Scaling and Resilience in Numerical Algorithms for Exascale Computing

THÈSE N$^{\text{O}}$ 8926 (2018)

PRÉSENTÉE LE 2 NOVEMBRE 2018
À LA FACULTÉ DES SCIENCES DE BASE
CHAIRE DE MATHÉMATIQUES COMPUTATIONNELLES ET SCIENCE DE LA SIMULATION
PROGRAMME DOCTORAL EN MATHÉMATIQUES

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Allan Svejstrup NIELSEN

acceptée sur proposition du jury:

Prof. F. Nobile, président du jury
Prof. J. S. Hesthaven, directeur de thèse
Prof. U. Rüde, rapporteur
Dr D. Ruprecht, rapporteur
Prof. M. Picasso, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2018

From Man or Angel the great Architect
Did wisely to conceal, and not divulge
His secrets, to be scan'd by them who ought
Rather admire; or, if they list to try
Conjecture, he his fabric of the heavens
Hath left to their disputes, perhaps to move
His laughter at their quaint opinions wide
Hereafter; ...

— John Milton, *Paradise Lost* (1667)

To my family…

Acknowledgements

I am profoundly grateful to my thesis advisor Prof. Jan S. Hesthaven, without his patience and guidance the thesis would not be. I would also like to thank Dr. Chris Cantwell, of the ICL Department of Aeronautics, for valuable discussions and contributions. In addition I would like to express my special appreciation and thanks to all the jury members on the thesis committee for taking the time to evaluate my work, as well as for their suggestions that greatly improved the quality of the thesis. Last but not the least, my sincere thanks goes out to all of the remarkably creative, smart, kind, diverse, fun, capable, and ambitious, students and researchers at the EPFL Chair of Computational Mathematics and Simulation Science, their support and encouragement have been outstanding.

Lausanne, August 8th 2018 A.S. Nielsen

Abstract

The first Petascale supercomputer, the IBM Roadrunner, went online in 2008. Ten years later, the community is now looking ahead to a new generation of Exascale machines. During the decade that has passed, several hundred Petascale capable machines have been installed worldwide, yet despite the abundance of machines, applications that scale to their full size remain rare. Large clusters now routinely have 50.000+ cores, some have several million. This extreme level of parallelism, that has allowed a theoretical compute capacity in excess of a million billion operations per second, turns out to be difficult to use in many applications of practical interest. Processors often end up spending more time waiting for synchronization, communication, and other coordinating operations to complete, rather than actually computing. Component reliability is another challenge facing HPC developers. If even a single processor fails, among many thousands, the user is forced to restart traditional applications, wasting valuable compute time. These issues collectively manifest themselves as low parallel efficiency, resulting in waste of energy and computational resources. Future performance improvements are expected to continue to come in large part due to increased parallelism. One may therefore speculate that the difficulties currently faced, when scaling applications to Petascale machines, will progressively worsen, making it difficult for scientists to harness the full potential of Exascale computing.

The thesis comprises two parts. Each part consists of several chapters discussing modifications of numerical algorithms to make them better suited for future Exascale machines. In the first part, the use of Parareal for Parallel-in-Time integration techniques for scalable numerical solution of partial differential equations is considered. We propose a new adaptive scheduler that optimize the parallel efficiency by minimizing the time-subdomain length without making communication of time-subdomains too costly. In conjunction with an appropriate preconditioner, we demonstrate that it is possible to obtain time-parallel speedup on the nonlinear shallow water equation, beyond what is possible using conventional spatial domain-decomposition techniques alone. The part is concluded with the proposal of a new method for constructing Parallel-in-Time integration schemes better suited for convection dominated problems.

In the second part, new ways of mitigating the impact of hardware failures are developed and presented. The topic is introduced with the creation of a new fault-tolerant variant of Parareal. In the chapter that follows, a C++ Library for multi-level checkpointing is presented. The library uses lightweight in-memory checkpoints, protected trough the use of erasure codes, to mitigate the impact of failures by decreasing the overhead of checkpointing and minimizing the compute work lost. Erasure codes have the unfortunate property that if more data blocks are lost than parity codes created, the data is effectively considered unrecoverable. The final chapter contains a preliminary study on partial information recovery for incomplete checksums. Under the assumption that some meta knowledge exists on the structure of the data encoded, we show that the data lost may be recovered, at least partially. This result is of interest not only in HPC but also in data centers where erasure codes are widely used to protect data efficiently.

Keywords – Exascale; Petascale; High-performance Computing; Parallel Computing; Parallel-in-time; Parareal; Hyperbolic PDEs; Shallow Water Equations; Resilience; Fault-tolerance

Zusammenfassung

Der erste Petascale Supercomputer, der IBM Roadrunner, ging 2008 ans Netz. Zehn Jahre später richtet sich der Blick der Gemeinschaft auf eine neue Generation von Exascale Maschinen. Im vergangenen Jahrzehnt wurden mehrere Hundert Petascale fähige Maschinen weltweit installiert. Trotz der Fülle Maschinen sind Anwendungen, welche das Potential vollumfänglich nutzen, weiterhin selten. Grosse Cluster haben heutzutage routinemässig 50’000+ Kerne, einige haben sogar mehrere Millionen. Das extreme Level der Parallelisierung, das theoretisch Rechenkapazitäten von über einer Million Milliarden Operationen pro Sekunde erlaubt, ist häufig zu schwierig zu implementieren, um für praktische Anwendungen von Nutzen zu sein. Prozessoren verwenden häufig mehr Zeit für das Abwarten der Synchronisation, Kommunikation und weiterer koordinativer Operationen, anstatt eigentliche Berechnungen durchzuführen. Die Verlässlichkeit der Komponenten stellt eine weitere Herausforderung für HPC Entwickler dar. Wenn nur ein einzelner Prozessor unter Tausenden einen Fehler begeht, muss der Benutzer die Applikation neu starten und verliert wertvolle Rechenzeit. Diese Schwierigkeiten manifestieren sich in einer niedrigen parallelen Effizienz, was zur Verschwendung von wertvoller Energie und Rechenzeit führt.

Diese Thesis ist zweiteilig. Jeder Teil besteht aus mehreren Kapiteln über die Modifikation von numerischen Algorithmen, um sie besser an die zukünftigen Exascale Maschinen anzupassen. Im ersten Teil wird die Benutzung von Parareal zu Parallel-in-Time Integration Techniken für das Lösen von skalierbaren, numerischen Lösungen von partiellen Differentialgleichungen betrachtet. Wir schlagen einen neuen adaptiven Scheduler vor, welcher die parallele Effizienz durch das Minimieren der Zeit Subdomain Länge optimiert, ohne die Kommunikation der Zeit Subdomains zu aufwendig zu gestalten. Wir zeigen, dass in Verbindung mit einem passenden Preconditioner möglich ist, time-parallel Speedup für die nonlineare Flachwassergleichung zu erreichen, welcher mit konventionellen spatial-decomposition Techniken nicht erreicht werden ann. Der Teil wird durch das Vorschlagen einer neuen Methode für das Konstruieren von Parallel-in-Time Integration Schemen abgerundet, welche besser für Konvektion dominierte Probleme geeignet ist.

Im zweiten Teil werden Techniken zum Reduzieren der Schäden von Hardwareausfällen entwickelt und präsentiert. Das Thema wird durch das Erstellen einer neuen ausfallresistenten Variante von Pareal begonnen. Im folgenden Kapitel wird eine C++

Bibliothek für multi-level Checkpoints präsentiert. Die Bibliothek verwendet leichte in-memory Checkpoints, welche, um die Folgen von Ausfällen zu reduzieren, durch Erasure Codes geschützt sind. Dies reduziert den Verlust an Rechenarbeit im Falle eines Ausfalles und die Overheadkosten durch Checkpointing. Wenn mehr Daten Blöcke verloren gehen, als Parity Codes erstellt wurden, haben Erasure Codes die Eigenschaft, dass die Daten als nicht wiederherstellbar angesehen werden. Das finale Kapitel beinhaltet eine vorläufige Studie über partielle Informationswiederherstellung für unvollständige Checksummen. Unter der Annahme, dass gewisses Metawissen über die Datenstruktur vorhanden ist, zeigen wir, dass die verlorenen Daten zumindest teilweise wiederhergestellt werden können. Dies könnte sich nicht nur für HPC als nützlich erweisen, sondern auch für Datencentren, welche häufig Eraser Codes verwenden um Daten zu schützen.

Keywords – Exascale; Petascale; Hochleistungsrechnen; Parallele Berechnung; Parallel-in-Zeit; Parareal; Hyperbolische PDEs; Flachwassergleichung; Fehlertoleranz

Résumé

L'IBM Roadrunner, le premier superordinateur de Petascale, a été mis en ligne en 2008. Dix ans plus tard, la communauté s'attend à une nouvelle génération de machines Exascale. Au cours de la décennie écoulée, des centaines de machines Petascale ont été installées dans le monde entier. Malgré l'abondance de ces machines, les applications qui s'adaptent à leur capacité maximale sont rares. De nos jours, les grandes clusters ont plus de 50 000 cœurs, même certaines en ont plusieurs millions. Le niveau extrême de parallelism qui permet une capacité de calcul théorique supérieure à un million de milliards d'opérations par seconde, se révèle difficile à utiliser dans de nombreuses applications d'intérêt. Souvent, les processeurs passent plus de temps à attendre que des processus comme la synchronisation, la communication et d'autres opérations de coordination se terminent, plutôt que de calculer. La fiabilité du composant est un autre défi auquel sont confrontés les développeurs HPC. Même si un seul processeur tombe en panne parmis plusieurs milliers, l'utilisateur est obligé de redémarrer l'application. L'amélioration des performances futures devrait continuer à se produire, en grande partie grâce au parallélisme. Il est donc supposé que les difficultés rencontrées actuellement lors de la mise à l'échelle des applications sur les machines Petascale se détérioreront progressivement, rendant difficile pour les scientifiques d'exploiter tout le potentiel de l'informatique Exascale.

Dans la première partie de cette thèse, nous considérons la solution numérique évolutive des équations différentielles partielles, par l'utilisation de Parareal pour les techniques d'intégration Parallel-in-Time. Nous proposons un nouveau programmateur adaptatif qui optimise l'efficacité du parallelism, en minimisant la longueur du domaine de temps, sans rendre la communication des sous-domaines de temps trop coûteuse. En conjonction avec un préconditionneur approprié, nous démontrons qu'il est possible d'accélérer parallèlement l'équation des eaux peu profondes non linéaires, en utilisant uniquement des techniques classiques de décomposition dans le domaine spatial. Nous concluons cette première partie en proposant une nouvelle méthode de construire des schémas d'intégration Parallel-in-Time.

Dans la seconde partie, de nouveaux moyens d'atténuer l'impact des défaillances du hardware sont développés et présentés. Le sujet est introduit par la proposition d'une nouvelle variante du fault-tolerant Parareal. Dans le chapitre qui suit, une bibliothèque C ++ pour le pointage à plusieurs niveaux est présentée. La bibliothèque utilise des points de contrôle légers en mémoire, protégés par l'implementation de codes d'effacement. Celà, en diminuant le travail de calcul perdu en cas d'échec, ainsi que la surcharge des points de contrôle. Effectivement, les codes d'éffacement considèrent comme irrécupérables les données si, plus de blocs de données sont perdus que des codes de parité sont créées. Le dernier chapitre contient une étude préliminaire sur la récupération partielle des informations pour les sommes de contrôle incomplètes. Nous montrons que les données perdues peuvent être au moins récupérées partiellement. Celà pourrait potentiellement être de pertinence, non seulement dans HPC, mais aussi dans les centres de données, où les codes d'effacement sont largement utilisés pour protéger les données d'une manière efficace.

***Keywords* –** Exascale ; Petascale ; Hochleistungsrechnen ; Parallele Berechnung ; Parallel-in-Zeit ; Parareal ; Hyperbolische PDEs ; Flachwassergleichung ; Fehlertoleranz

Contents

Acknowledgements **vii**

Abstract (English/Français/Deutsch) **ix**

List of figures **xvii**

List of tables **xxxiii**

I Introduction **1**

1 A Brief History of Computing **3**

2 Frontiers of Computational Science and Engineering **9**

3 Moving to Exascale: Status and Open Challenges **13**
3.1 Motivation for Exascale 13
3.2 Planned Exascale Supercomputing Facilities 16
3.3 Exascale Challenges on Exascale Machines 19

4 Thesis Content and Contributions **25**
4.1 Essential Concepts 25
4.2 Outline of Thesis 27

II Parallel-in-Time Integration for the Scalable Solution of PDEs **29**

5 Introduction to Parallel-in-Time Integration Techniques **31**
5.1 Parallel-in-Time Integration for PDEs 32
5.2 The Parareal Method 34
5.3 Contributions in Part II 37

6 Numerical Experiments: Parareal Applied to Hyperbolic Problems **39**
6.1 The impact of Phase and Amplitude 40
6.2 An Illustration of the Correction Procedure 45
6.3 Summary 49

7 Communication Aware Adaptive Parareal **51**

7.1 Introduction 52
7.2 2D Shallow Water Equation with Explicit SSP-RK WENO 52
7.2.1 Introducing the Operators 53
7.2.2 Parareal Convergence 57
7.2.3 Introducing the Test Case 63
7.3 Space-Time Domain Decomposition 67
7.3.1 CAAP Convergence 76
7.4 Optimizing Time-Subdomain Length 78
7.4.1 Single Cycle Analysis 79
7.4.2 Multi Cycle Analysis 81
7.5 Numerical Experiments: Parallel Scaling 85
7.6 Summary 91

8 Time-Parallel Integration for Convection Dominated Problems **93**

8.1 Modifying Parareal 93
8.1.1 Fourier Space Phase Correction 93
8.1.2 Wave-Local Phase Correction 95
8.2 Deriving Parallel-in-Time Integration Schemes 99
8.2.1 An Exact Correction Procedure 101
8.2.2 A New 2-Level Scheme 103
8.2.3 Numerical Experiments 107
8.3 Summary 109

III Fault-tolerant Algorithms for Exascale Systems **111**

9 An Introduction to HPC Resilience and Fault-tolerance **113**

9.1 Classification of Errors and Failures 114
9.2 Strategies for Detection and Repair 118
9.3 Contributions in Part III 121

10 Fault-Tolerance in Parareal **123**

10.1 Introduction 123
10.1.1 Parallel-in-Time using Parareal 124
10.1.2 Resilience and Fault-tolerance 124
10.2 Recovering from Node-Loss 125
10.2.1 A Fault-Tolerant Scheduler 127
10.2.2 Failure Analysis 132
10.3 Guarding against SDC Errors 136
10.3.1 SDC Resilient Parareal 136
10.3.2 Numerical Experiments 137
10.4 Summary 139

11 Llama: A new library for Multilevel Checkpointing 141
11.1 Checkpoint-Restart for HPC Fault-Tolerance 143
    11.1.1 Checkpointing for Resilience 143
    11.1.2 Libraries for Automatic Checkpoint-Restart 145
    11.1.3 User Level Failure Mitigation MPI 147
11.2 The Llama Library 147
    11.2.1 Design 147
    11.2.2 Usage 151
11.3 Numerical Experiments 158
    11.3.1 Creating the Llama Guard 159
    11.3.2 Adding a Checkpoint to the Guard 160
    11.3.3 Cost of Updating Checkpoints 163
    11.3.4 Cost of Data Recovery 166
11.4 Summary 170

12 Partial Information Recovery with Incomplete Checksums 171
12.1 The Weighted Checksum Scheme 173
12.2 Partial Information Recovery for Incomplete Checksums 175
    12.2.1 Minimizing Distance to Inexact Data 176
    12.2.2 An Iterative Recovery Scheme 180
12.3 Summary 184

IV Summary 187
13.4 Novel Contributions 189
13.5 Outlook and Perspectives 190

Appendix 197

A CAAP Scheduler on Test Eq. 199

B CAAP Convergence on Test Eq. 205

C CAAP Subdomain length for Optimal Speedup 215

D CAAP Scheduler on 2D SHW 223

E Llama Library Example Code 229

F Llama Library Scaling Tests 235

Bibliography 240

Curriculum Vitae 269

List of Figures

2.1 List of major landmark computations in scientific computing on top supercomputers. The list is not meant to be exhaustive, and only results for where a cite-able source, with details on the study, could be found, have been included. The * indicates that an associated paper was awarded the Gordon Bell prize in either the Performance or the Scaling category. Most entries of dates have been moved slightly left or right so to allow for readable labels. 12

3.1 The average energy consumption and average compute efficiency of the 50 most powerful supercomputers in the world as reported on the TOP500 list[209]. Tracking power usage was only introduced in 2008 and compute efficiency in 2011 and data was not available for every machine on the list for all years. The data points are therefore computed as averages of the TOP50 machines for which data was available in a given year. 19

3.2 Average CPU frequency of the 50 most powerful machines on the TOP500 list and of all machines on the list. The frequency is averaged over the main processor, i.e, not including coprocessors. Figure compiled using freely available data from TOP500 lists dating back to 1998 [209]. . . . 20

3.3 Average compute performance, number of cores and sockets of the (a) 50 and (b) 500 most powerful machines on the TOP500 list as a function of time. Main CPU cores indicate the average number of cores of the main architecture, used on each machine. Main CPU + Acc. cores indicate the average total number of cores, i.e. including any coprocessor. The dashed line on the socket-numbers between 2005 and 2010 indicates estimates as seperate tracking of cores per socket was not included in the data until 2010 even though multi-core main processors appeared as early as 2005. The two figures were compiled using freely available data from TOP500 lists dating back to 1998 [209]. 24

6.1 The application of Parareal for parallel-in-time integration of the advection-diffusion equation with $\kappa = 10^{-5}$ and $a = 1$ to $T = 2.5$ using the initial condition in figure (a). The dispersive operators $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ are created using 301 and 31 points in space respectively, with a 4th order compact finite difference scheme. The dissipative operators $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ were created using a 1st order upwind scheme for the advection term and centered finite difference for the diffusion term, applied to a mesh with 601 and 61 points in space respectively. 41

6.2 Convergence when computing an approximation to the solution of (6.1) with $\kappa = 10^{-3}$ and $a = 1$ to $T = 10$ on $n_t = N = 50$ time-subdomains. (a,c,e) using a factor 6 coarsened space-time grid as a dispersive $\mathcal{G}_{\Delta T}$. (b,d,f) using a lower order operator as a dissipative $\mathcal{G}_{\Delta T}$. In both cases $\mathcal{F}_{\Delta T}$ is the 4th order dispersive scheme. 43

6.3 Convergence when computing an approximation to the solution of (6.1) with $\kappa = 10^{-5}$ and $a = 1$ to $T = 10$ on $n_t = 50$ time-subdomains. (a,c,e) using a factor 6 coarsened space-time grid as a dispersive $\mathcal{G}_{\Delta T}$. (b,d,f) using a lower order operator as a dissipative $\mathcal{G}_{\Delta T}$. In both cases $\mathcal{F}_{\Delta T}$ is the 4th order dispersive scheme. 44

6.4 Illustration of the Parareal correction procedure on a shock moving to the right with a constant velocity of two steps per ΔT. $\mathcal{F}_{\Delta T}$ is exact in that it takes any state U and moves it two steps to the right. The coarse operator $\mathcal{G}_{\Delta T}$ also moves the solution two steps to the right, but introduces an error on the amplitude such that $\mathcal{G}_{\Delta T} U_n^k = \mathcal{G}_{\Delta T} U_n^k - 0.25$ if $\mathcal{G}_{\Delta T} U_n^k \neq 0$ and $\mathcal{G}_{\Delta T} U_n^k = \mathcal{G}_{\Delta T} U_n^k$ otherwise. One notes that U_n^k for all n converge to the exact solution at $k = 1$. 46

6.5 Illustration of the Parareal correction procedure on a shock moving to the right with a constant velocity of two steps per ΔT. $\mathcal{F}_{\Delta T}$ is exact in that it takes any state U and moves it two steps to the right. The coarse operator $\mathcal{G}_{\Delta T}$ introduces an error as it only moves a state U one step to the right for each ΔT. One notes that large oscillations are introduced around the shock and that no meaningful correction is made on the state U_n^k for all n such that $n > k$. 47

6.6 Illustration of the Parareal correction procedure on a wave with sharp edges moving to the right with a constant velocity of two steps per ΔT. $\mathcal{F}_{\Delta T}$ is exact in that it takes any state U and moves it two steps to the right. The coarse operator $\mathcal{G}_{\Delta T}$ introduces an error as it only moves a state U one step to the right for each ΔT. One notes that for all points x_i for which it is true that $\frac{\partial}{\partial x}\mathcal{F}_{\Delta T} U_n^k(x)\big|_{x_i} = \frac{\partial}{\partial x}\mathcal{G}_{\Delta T} U_n^k(x)\big|_{x_i} = \frac{\partial}{\partial x}\mathcal{G}_{\Delta T} U_n^{k+1}(x)\big|_{x_i}$, the correction is exact already by the first correction $k = 1$. For the points x_i at which the angles differ, large oscillations are introduced and no meaningful correction made. 48

7.1 Initial condition (7.26) with periodic boundary conditions. The reference solution is computed using a cell size $dx = 10^{-4}$. (a) $T = 0$. (b) $T = 0.1$. (c) $T = 1$. 59

7.2 ϵ_u and $\epsilon_{\mathcal{F}}$ as a function of k for the Parareal algorithm when solving the 1D non-linear shallow water wave equation on $N = 20$ time-subdomains using the fine scheme and Parareal preconditioner, described in section 7.2.1. $\epsilon_W u$ and $\epsilon_{\mathcal{F}}$ are defined in (7.24) and (7.25). The red line indicates the error of the fine solution with respect to the true solution. The red dotted line indicates the error of the preconditioner with respect to the true solution. 60

7.3 ϵ_u and $\epsilon_{\mathcal{F}}$ as a function of k for the Parareal algorithm when solving the 1D non-linear shallow water wave equation on $N = 50$ time-subdomains using the fine scheme and Parareal preconditioner described in section 7.2.1. ϵ_u and $\epsilon_{\mathcal{F}}$ are defined in (7.24) and (7.25). The red line indicates the error of the fine solution with respect to the true solution. The red dotted line indicated the error of the preconditioner with respect to the true solution. 61

7.4 Solving the inhomogeneous two-dimensional shallow water wave equation (7.1) with absorbing boundary conditions and a thin-layer mesh reduction technique for land inundation. The equation is solved on a structured rectangular mesh spanning a 400km by 400km map centered over the O'Ahu and Hawaii islands in the pacific. The bathymetry and topography was taken from the free ETOPO1 global relief data-set[4]. The initial conditions at $T = T_0$ consists of multiple superimposed waves rising above the steady-state solution. A red-black-purple coloring scheme is used to indicate the deviation of water height with respect to steady state water-line over time. 62

7.5 The initial condition, (7.29) for a wave amplitude of $A = 500m$, a frequency $\omega = 4$ and width a $\sigma = 10km$ with the bathymetry (7.27) using $L = 1000km$, a basin radius of $a = 400km$, and a depth $h_0 = 1km$. The solution of (7.1) with the initial condition depicted, is given for 10 minute intervals in Fig. 7.6 and 7.7. 64

7.6 Water height as a function of time for the test case (7.29) at 10 minute intervals from $T_0 = 0$ to $T = 50$min. Beige and green colors are used to indicate land whilst shades of blue indicate water depth. Light effects have been added to highlight the location of shocks. 65

7.7 Water height as a function of time for the test case (7.29) at 10 minute intervals from $T = 60$ to $T = 110$min. Beige and green colors are used to indicate land whilst shades of blue indicate water depth. Light effects have been added to highlight the location of shocks. 66

7.8 Profiling the parallel-in-space WENO based Tsunami simulation when solving the 1600x1600 cells test-case presented in Section 7.2.3. As the number of cores increase, the cost of the halo-exchange between subdomains comes to dominate. Due to the small size of the test-case, the code effectively stops scaling with 128 cores. The code was profiled on the EPFL Bellatrix cluster. Each node in the cluster contains two 8-core Intel Xeon E5-2660 CPUs and Inifiniband QDR 2:1 connectivity between nodes. Using a single core on a single node, the computation takes 11817 seconds to complete. Using 4 nodes for a total of 64 cores, the computation needs 278 seconds to complete. 67

7.9 Domain decomposition in space and time of the WENO based solver introduced in Section 7.2. White lines indicate the division of the spatial domains, and each image indicate a subdomain spanning ΔT in time. 68

7.10 The standard fully distributed Parareal[9]. Each time-subdomain is handled by a unique node-group, possibly parallel-in-space. Dark gray indicates that a node-group is computing the preconditioner $\mathcal{G}_{\Delta T}$, light gray indicates that a node-group is computing $\mathcal{F}_{\Delta T}$. Drawn for $n_t = 8$. Shorter time-subdomains may lead to faster convergence, but this comes at the cost of more frequent communication of the solution-states at time-subdomain interfaces. Drawn as if the tolerance was satisfied by the 3rd correction. 69

7.11 Multiple consecutive executions of the standard fully distributed Parareal[9]. Each time-subdomain is handled by a unique node-group, possibly parallel-in-space. Dark gray indicates that a node-group is computing the preconditioner $\mathcal{G}_{\Delta T}$, light gray indicates that a node-group is computing $\mathcal{F}_{\Delta T}$. Drawn for $n_t = 8$, $n_c = 3$. Shorter time-subdomains may lead to faster convergence, but this comes at the cost of more frequent communication of the solution-states at time-subdomain interfaces. In each cycle, the tolerance is satisfied by the 3rd correction. The direction of information flow shifts with each cycle to reduce the number of time-subdomain interface states sent. 69

7.12 Schematic representation of a proposed Adaptive Parareal scheduler. The scheduler lets multiple cycles of Parareal overlap during execution. Drawn here for $n_t = 8$, $n_c = 3$. Dark gray indicates that a node-group is computing the preconditioner $\mathcal{G}_{\Delta T}$, light gray indicates that a node-group is computing $\mathcal{F}_{\Delta T}$. Black dots and arrows indicate sending and receiving of time-subdomain interface solution state. Blue arrows indicate a signal being sent to inform a node-group working on a time-subdomain, that the next time-subdomain has become active. Each node-group has a boolean flag that indicates if the next time-subdomain is active or not. (a) $\beta = 0$ for a fully patient model in which node-groups that finished a time-subdomain in a cycle will wait for a correction to be made before receiving a new state to commence their work. (b) $\beta = 1$ for a fully impatient model in which node-groups that finished a time-subdomain in a cycle will receive a new state to commence their work immediately. 70

7.13 Error of U_n^k with respect to the analytic solution to (7.30) as a function of t for parallel-in-time integration of (7.1) with time-steps and tolerance $\epsilon = 0.01$, used for the measurements in Figure 7.14. (a) Standard Parareal, $n_c = 1$. (b) Standard Parareal, $n_c = 5$. (c) Adaptive Parareal, $n_c = 5$, $\beta = 0$. (d) Adaptive Parareal, $n_c = 5$, $\beta = 0.8$. The black dashed line indicates accuracy of the fine operator with respect to the analytical solution, and the black dotted line indicates the accuracy of $U_n^{k_{conv}}$ obtained at the iteration for which the convergence criteria was satisfied. 77

7.14 Parallel speedup and efficiency measured when solving (7.30) as a function of time-subdomain length ΔT (a)-(b), and number of cycles n_c (c)-(d). Using $n_t = 20$ processors parallel-in-time and letting $C_{\mathcal{F}} = 10000$ms, $C_{\mathcal{G}} = 100$ms, $T_c^w = 100$ms. The square, circle and triangle markers indicate the estimates for the optimal time-subdomain length using (7.44), (7.51) and the average of the two, respectively. 84

7.15 Water surface at $T = 60$min starting with the initial condition described in Section 7.2.3 at $T = 0$min as computed when using (a) Roe's method (b) WENO SSP-ERK (c) CAAP with tolerance $\epsilon = 10^{-4}$ on the norm of the difference between two consecutive iterations. The coarse operator captures wave speeds very well but introduces dissipative errors. From a bird-eye perspective it appears that CAAP arrives at the same solution as the parallel WENO SSP-ERK with sequential time-stepping. In Figure 7.16 a zoomed in view of the profile around a shock is presented. Units in kilometers. 86

7.16 Cross section of the water height at $y = 500km$ where two shocks meet, see figure 7.15. The cell width is $dx = 0.625$km, so the above profile is resolved with 24 cells. The error of the Parareal solution is largest in magnitude around shocks. The time-to-solution using the fine and the coarse solver on a single node, sequentially in time, is respectively 912s and 79s. For CAAP, using a single node in space on each of 16 simultaneously active time-subdomains, the time-to-solution is 205s, 199s, 242s and 361s respectively. It is somewhat surprising that for the largest tolerance we do not measure the shortest time-to-solution. Looking at the log files of the simulation, it appears that this is due to less effective load balancing and that in some intervals convergence happens in just 1 iteration, i.e. second correction, whilst in other time-subdomains it takes several iterations despite the large tolerance. . . . 88

7.17 Speedup as a function of number of cores used in the space-time parallel code for solving the SHW test case introduced in Section 7.2.3 as measured on the 424-node EPFL Bellatrix cluster. $T_{end} = 60$min, and $n_c = \{20, 15, 15\}$ for $n_t = \{8, 16, 24\}$ respectively. Tolerance $\epsilon = 10^{-4}$ was used as the convergence criteria. 90

8.1 The application of Parareal for parallel-in-time integration, on $n_t = 50$ time-subdomains, of the advection-diffusion equation with $\kappa = 10^{-5}$ and $a = 1$ to $T = 2.5$ using the initial condition in figure 6.1a. The dispersive operators $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ are created using 301 and 61 points in space respectively, with a 4th order compact finite difference scheme. Figures (a) and (b) shows the sequential solution $\mathcal{F}_{\Delta T}^{N=50}u_0$, the true solution $u(t = 2.5)$, and consecutive iterations $U_k^{N=50}$. In figure (a), correction is performed in Fourier space as outlined in section 8.1.1, in figure (b) regular Parareal corrections are used. (c) shows the error at $t = 2.5$ as measured with respect to the true solution. (d) shows the error as measured with respect to the fine solution. All errors are measured in the infinity norm on the entire space-time domain. 94

8.2 Convergence of the original Parareal method, and the modified variant, when computing an approximation to the solution of (6.1) with $\kappa = 10^{-3}$ and $a = 1$ to $T = 10$ on $n_t = 20$ time-subdomains using a factor 6 coarsened space-time grid as $\mathcal{G}_{\Delta T}$. (a) show iterations at $T = 10$ of the original method (b) shows iterations of the modified variant. (c) error in infinity norm measured with respect to the true solution $u(T = 10)$ and (d) error in infinity norm measured with respect to the sequential fine solution at $u(T = 10)$. 97

8.3 Convergence of the original Parareal method, and the modified variant, when computing an approximation to the solution of (6.1) with $\kappa = 10^{-5}$ and $a = 1$ to $T = 10$ on $n_t = 20$ time-subdomains using a factor 6 coarsened space-time grid as $\mathcal{G}_{\Delta T}$. (a) show iterations at $T = 10$ of the original method (b) shows iterations of the modified variant. (c) error in infinity norm measured with respect to the true solution $u(T = 10)$ and (d) error in infinity norm measured with respect to the sequential fine solution at $u(T = 10)$. 98

8.4 Iterative time-parallel solution of the advection equation, $\alpha = 1$, solved using the upwind scheme with CFL = 0.5 for $\mathcal{F}_{\cdot\mathcal{T}}$. The dissipative $\mathcal{G}_{\Delta T}$ was created by adding a diffusion term with $\kappa = 5.0 \cdot 10^3$ to $\mathcal{F}_{\cdot\mathcal{T}}$. The dispersive $\mathcal{G}_{\Delta T}$ was created by letting $\alpha = 1.05$. In (a)(c), error was measured with respect to the true solution. In (b)(d), error was measured with respect to the sequential fine solution. Parareal was used in figures (a)(b). In figure (c)(d), the correction procedure was performed as described in Section 8.2.1. Integration until $T = 2.5$ on $n_t = 50$ time-subdomains, starting from the discontinuous initial condition depicted in Figure 6.1a. $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ were constructed using 101 points in space with CFL $= 0.5$. 102

8.5 Convergence of two parallel-in-time integration methods when applied to solve the test equation (8.32) as outlined in section 8.2.3. (a) Error measured with respect to the true solution. (b) Error measured with respect to the sequential solution. 105

8.6 Convergence of two parallel-in-time integration methods when applied to solve the advection-diffusion equation (6.1) as outlined in section 8.2.3. (a) Error measured with respect to the true solution. (b) Error measured with respect to the sequential fine solution. 107

8.7 Convergence of two parallel-in-time integration methods when applied to solve the advection-diffusion equation (6.1) as outlined in section 8.2.3 with (a)-(b) smooth initial condition, and (c)-(d) discontinuous initial condition. $\kappa = 10^{-5}$, $a = 1$. Integration until $T = 2.5$ on $n_t = 50$ time-subdomains. $\mathcal{F}_{\Delta T}$ 201 points in space and $\mathcal{G}_{\Delta T}$ 21 points in space. (a)(c) Error measured with respect to the true solution. (b)(d) Error measured with respect to the sequential fine solution. 108

10.1 The “Fully-Distributed” work scheduling of the Parareal algorithm as proposed in [9]. Light grey boxes indicate a node-group for computing $\mathcal{F}_{\Delta T}\mathbf{U}$ and dark gray indicates that a node group is evaluating the coarse operator $\mathcal{G}_{\Delta T}\mathbf{U}$. Drawn for $N = 6$ time-subdomains and convergence in $k_{conv} = 3$ iterations. 126

10.2 Schematic visualization of the recovery procedure of the Fault-Tolerant algorithm with $N = 6$ time-subdomains and fault-free convergence in $k_{conv} = 3$ iterations. Failed node-groups injected at $\{id_{\Delta T}, k_{err}\} = \{3,2\}, \{4,2\}$. The north east line pattern indicates failed nodes that no longer participate. 126

10.3 Execution time in seconds for the unprotected algorithm, and for the proposed fault-tolerant algorithm with one or multiple node-group losses located at $\{id_{\Delta T}, k_{err}\}$. (a) No errors. (b) $\{15,1\}$. (c) $\{15,3\}$. (d) $\{4,1\}, \{5,2\}$. (e) $\{11,2\}, \{12,2\}, \{15,3\}$. The number in parenthesis indicates iterations to convergence. 16 ranks were used in space on 16 time-subdomains for 256 ranks in total. 132

10.4 A tree indicating all possible routes to successful execution as drawn for a small problem with $N = 4$ time-subdomains and failure-free convergence in $k_{conv} = 2$ iterations. The probability of successful execution $\mathcal{P}_n^{N,k}$ at a given node may be computed using (10.4). l_1 and l_2 are shown for each iteration k on the right as computed using (10.2) and (10.3). The blue circle indicates successful execution of the algorithm. Branches leading to failed execution are not drawn. 134

10.5 Probability of successful execution for the unprotected algorithm and the Fault-Tolerant variant $N = 16$, $r = 32$, $k_{conv} = 2$ and failure rate $\mu_{\Delta T}^{\mathcal{G}} = 0.0001$. The dashed line indicate the proportion of errors in the unprotected algorithm that the Fault-Tolerant version may recover from. 135

10.6 Convergence rate when the solution procedure is subjected to silent-data corruption (SDCs). The unmarked solid black line indicate the convergence rate for the error-free solution procedure. (a) Average error. (b) Error variance. 138

11.1 Dependency graph of the Llama header that must be included in applications using the library. The class declaration for the two checkpoint types supported are written disk.h and memory.h. They both derive from a pure virtual class *checkpointinterface* in the interface header. The interface header also includes the decleration of the Guard class. The testing header includes functionality used in the ctest package which includes 30 tests, testing to verify correct execution of the various components of Llama and their interactions. 148

11.2 Scaling test, time to create Llama Guard as a function of the number of cores as tested on the EPFL Fidis cluster. The time to create the checkpoint increases slightly with an increasing number of cores, but varies mostly due to difference in network load and job placement on the shared general purpose cluster. 159

11.3 Scaling test, time to attach a checkpoint-type to the Llama Guard as a function of number of cores. The time increases slightly with an increasing number of cores, the overhead of checking parameters and creating a matrix of local communicators makes the in-memory type checkpoint slower to initialize. 160

11.4 The wall-time cost of adding an array to the Llama Guard as measured on the EPFL Fidis general purpose cluster for 112 to 7168 cores. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times to illustrate the variance in timings due to job-placement and network load when using a shared cluster. The cost of adding an array is observed to be almost constant, only vaguely increasing with an increasing number of cores. 162

11.5 Weak scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints. 164

11.6 Strong scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding four parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system. 165

11.7 Weak scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints. 168

11.8 Strong scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding 4 parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system. 169

12.1 Numerical experiment testing the method (12.9) for partial information recovery in incomplete checksums. The data indicated by the black line was stored in $k = 100$ separate containers. $m = 15$ checksum vectors were created to protect the content of the containers. The $k_{lost} = 20$ first containers are removed, i.e. 33% more data vectors are lost than checksum code vectors created. 178

12.2 Numerical experiment testing the method (12.9) for partial information recovery in incomplete checksums. The data indicated by the black line was stored in $k = 100$ separate containers. $m = 18$ checksum vectors were created to protect the content of the containers. The $k_{lost} = 20$ first containers are removed, i.e. 11% more data vectors are lost than checksum code vectors created. 179

12.3 The original version of the 16 images used to demonstrate the method outlined. Each image is 512x512 pixels, and stored in separate data containers that are assumed to be failure prone. The images are taken from the The USC-SIPI Image Database[303]. In figures 12.4 and 12.5, the partial information recovery procedure is demonstrated when removing 4 and 6 images respectively. 181

12.4 Three checksums were computed to protect the content of the 16 containers. (a) Shows four images removed. (b) Depicts the recovered images. 182

12.5 Three checksums were computed to protect the content of the 16 containers. (a) Shows six images removed. (b) Depicts the recovered images. 183

13.6 Number of system and flop/s performance share of processors among the systems recorded on the TOP500 list updated twice a year since 1993[209]. The list of CPU's in table 13.1 was used to distinguish between commodity and non-commodity hardware. Note that the share of systems, not relying only on commodity CPU's, appears to be increasing. 193

13.7 Number of system and flop/s performance share of processors among the 50 most powerfull systems as recorded on the TOP500 list updated twice a year since 1993[209]. The list of CPU's in Table 13.1 was used to distinguish between commodity and non-commodity hardware. Note that machines using server/HPC specific CPUs and those using co-processors, now has a combined 80% performance share among the 50 most powerful systems. 194

A.1 "Fully distributed" Parareal. $n_t = 20$, $n_c = 1$, $T = 100$, $\Delta T = 5$, $dT = 10^{-3}$, , $dt = 10^{-5}$, $T_{\mathcal{F}} = 5000ms$, $T_{\mathcal{G}} = 50ms$, $T_{\mathcal{C}} = 2ms$. Blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed receive of solution state and convergence flag. Dark gray indicates $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$ computation, light gray indicates $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$. Measured speedup 3.19, parallel efficiency 15.9%. 200

A.2 Multiple consecutive executions of the standard "fully distributed" Parareal. $n_t = 20$, $n_c = 6$, $T = 100$, $\Delta T = 0.83\bar{3}$, $dT = 10^{-3}$, $dt = 10^{-5}$, $T_{\mathcal{F}} = 4167ms$, $T_{\mathcal{G}} = 42ms$, $T_{\mathcal{C}} = 2ms$. Blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed recieve of solution state and convergence flag. Dark gray computation of $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$. Light gray, and light gray with lines, computation of $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$. Measured speedup 6.04, parallel efficiency 30.2%. 201

A.3 Adaptive Parareal Scheduler. $\beta = 0$, $n_t = 20$, $n_c = 6$, $T = 100$, $\Delta T = 0.83\bar{3}$, $dT = 10^{-3}$, $dt = 10^{-5}$, $T_{\mathcal{F}} = 4167ms$, $T_{\mathcal{G}} = 42ms$, $T_{\mathcal{C}} = 2ms$. Blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed receive of solution state and convergence flag. Green square and arrow indicates signal flag. Dark gray computation of $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$. Light gray, and light gray with lines, computation of $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$. Measured speedup 7.73, parallel efficiency 37.4%. . . . 202

A.4 Adaptive Parareal Scheduler. $\beta = 1$, $n_t = 20$, $n_c = 6$, $T = 100$, $\Delta T = 0.83\bar{3}$, $dT = 10^{-3}$, $dt = 10^{-5}$, $T_{\mathcal{F}} = 4167ms$, $T_{\mathcal{G}} = 42ms$, $T_{\mathcal{C}} = 2ms$. Blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed receive of solution state and convergence flag. Green square and arrow indicates signal flag. Dark gray computation of $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$. Light gray, and light gray with lines, computation of $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$. Measured speedup 7.73, parallel efficiency 38.6%. 203

B.1 Error as a function of simulation time when using standard single-cycle Parareal to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 10$ is performed across $N_c = 1$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-3}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f). 206

B.2 Error as a function of simulation time when using standard single-cycle Parareal to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 100$ is performed across $N_c = 1$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-2}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f). 207

B.3 Error as a function of simulation time when using standard multiple-cycle Parareal to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 10$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-3}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f). 208

B.4 Error as a function of simulation time when using standard multiple-cycle Parareal to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 100$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-2}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f). 209

B.5 Error as a function of simulation time when using Adaptive Parareal with $\beta = 0$ to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 10$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-3}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f). 210

B.6 Error as a function of simulation time when using Adaptive Parareal with $\beta = 0$ to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 100$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-2}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f). 211

B.7 Error as a function of simulation time when using Adaptive Parareal with $\beta = 0.8$ to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 10$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-3}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f). 212

B.8 Error as a function of simulation time when using Adaptive Parareal with $\beta = 0.8$ to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 100$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-2}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f). 213

C.1 Parallel speedup as a function of time-subdomain length δt when integrating equation 7.30 to $T = 100$ using multiple consecutive cycles of a standard fully distributed parareal, and when using CAAP. Circles indicate a priori estimated optimal δT using equation (7.44) and equation (7.51). $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 1ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$. 216

C.2 Parallel efficiency as a function of number of cycles n_c used in the time-parallel integration to $T = 100$. The black line represents data measured using multiple consecutive cycles of a standard fully distributed parareal, and the yellow/purple dashed lines are measured using CAAP. Circles indicate a priori estimated optimal δT using equation (7.44) and equation (7.51). $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 1ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$. 217

C.3 Parallel speedup as a function of timesubdomain length δt when integrating equation 7.30 to $T = 100$ using multiple consecutive cycles of a standard fully distributed parareal scheduler, and when using CAAP. Circles indicate a priori estimated optimal δT using equation 7.44 and 7.51. $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 10ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$. 218

C.4 Parallel speedup as a function of number of cycles n_c used in the time-parallel integration to $T = 100$. The black line represents data measured using multiple consecutive cycles of a standard fully distributed parareal, and the yellow/purple dashed lines CAAP. Circles indicate a priori estimated optimal δT using equation 7.44 and 7.51. $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 10ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$. 219

C.5 Parallel speedup as a function of timesubdomain length δt when integrating equation 7.30 to $T = 100$ using multiple consecutive cycles of a standard fully distributed parareal, and when using CAAP. Circles indicate a priori estimated optimal δT using equation 7.44 and 7.51. $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 100ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$. 220

C.6 Parallel speedup as a function of number of cycles n_c used in the time-parallel integration to $T = 100$. The black line represents data measured using multiple consecutive cycles of a standard fully distributed parareal, and the yellow/purple dashed lines are measured using CAAP. Circles indicate a priori estimated optimal δT using equation 7.44 and 7.51. $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 100ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$. 221

D.1 Fully distributed Parareal applied to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 1$. Integration to $T = 100$min, 1 node in space. 224

D.2 Multiple consecutive executions of the standard “fully distributed” Parareal to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 10$. Integration to $T = 100$min, 1 node in space. 225

D.3 Adaptive Parareal with $\beta = 0.0$ used to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 10$. Integration to $T = 100$min, 1 node in space. 226

D.4 Adaptive Parareal with $\beta = 0.4$ used to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 10$. Integration to $T = 100$min, 1 node in space. 227

D.5 Adaptive Parareal with $\beta = 0.8$ used to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 10$. Integration to $T = 100$min, 1 node in space. 228

F.1 Weak scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints. 236

F.2 Strong scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding 4 parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system. . . . 237

F.3 Weak scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints. 238

F.4 Strong scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding 4 parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system. 239

List of Tables

7.1 Parallel acceleration as measured using $n_t = \{8, 16, 24\}$ simultaneously active time-subdomains. Within each time-subdomain, 5 nodes (80 cores) are used parallel in space, for a total of $\{640, 1280, 1920\}$ cores, respectively. The first parallel efficiency column indicates the efficiency of the parallel-in-time integration procedure, the second column indicates the combined space-time parallel efficiency. 89

13.1 List of processors used in HPC from 1993 to 2018 which was considered commodity processors when creating figures 13.6 and 13.7. The distinction between commodity and non-commodity hardware is not sharp. The above processors were selected because they were all, at some point, used in personal computer workstations, or it was planned for them to be used for such purposes during the design stage. 192

Introduction Part I

1 A Brief History of Computing

A modest modern day computer easily calculates at a rate of several billion arithmetic operations per second. Computers have become an ubiquitous part of most aspects of society and it is hard to imagine computational science without them. Computational science may however be said to predate the modern day computer by centuries and traces its roots to the early days of computing the movement of planets and comets by hand.

In the late summer of 1695, the English astronomer Edmund Halley (1656–1742) was working on validating statements made by Issac Newton (1642–1727) on the nature of comets. At the time, calculus was a new invention that had been developed in part to explain the motion of planets by physical laws rather than that of the actions of superhuman beings. Newton had theorized that the movement of two objects under the influence of a single universal force, gravity, are bound to follow certain paths: Parabola, hyperbola or that of the cyclical ellipse[226]. At the time, comets were thought of as mysterious visitors that appeared at irregular intervals, no obvious explanation was known for their coming and going[142]. Newton had postulated that comets might be celestial objects that came from distant points in the universe and moved in a tight parabola around the sun before returning to the void from which they came.

Astronomers, fortunate enough to observe a comet, would measure and record their position against fixed stars as they moved across the sky. Halley had taken it upon himself to investigate Newtons explanation by attempting to fit parabola curves to the paths of previously recorded comets. Halley found that some comets did indeed appear to follow a parabola curve around the sun. However, more importantly, he noticed that three comets in the records appeared to follow the very same path across the sky. One recorded in 1531, another in 1607 and finally one recorded by himself in 1682. Halley decided to examine if an elliptical orbit would fit the path of the three earlier observations, and this turned out to indeed be the case. Confidently he proclaimed, first to Newton and later to the whole astronomical community, that the three comets observed must have been the same returning celestial object[151, 46]. The claim was met with some skepticism from the community. His analysis suggested that the comet should have a fixed period of return, yet the observations clearly showed that the return did not occur at an exact period of 75 years. Halley suggested that the irregular period was due to the influence of the gravitational pull of Saturn and Jupiter but to his dismay even Newton was unable to find a simple computable expression that described the motion of such a system.

The Sun, Jupiter, and Saturn form a three body system, each exerting an influence upon the other two, a problem for which Newton's calculus could not provide a simple solution. Edmund Halley made a last prediction in a paper published after his death in 1742. He estimated that the effect of Saturn and Jupiter would delay the arrival by 1-2 years thus predicting the return "about the end of the year 1758, or the beginning of the next"[152]. In the years that followed, the anticipated return of Halley's comet came to be considered as a test of Newtons theory of gravitation. Despite the attention the problem received from the public, for more than a decade nobody attempted a complete treatment of the gravitational impact of Jupiter and Saturn to compute an exact date of return[142]. This can perhaps be attributed to the large amount of calculations needed or perhaps due to a realization that correctly solving the problem would require new computational techniques, going beyond those Halley had used for his estimates.

In fact only one attempt was made. A French mathematician by the name of Alex-Claude Clairaut (1713–1765) had recently developed a computational approach for handling the three body problem and was looking for a grand test-case to demonstrate his method and propel him into fame. Clairaut's method consisted of dividing the movements of Saturn and Jupiter as they revolved around the sun into tiny steps of just a degree or so. In each step he would first advance the planets by a small step in time along the idealized path of the ellipse, which would be followed by the computation of an adjustment to each of the ellipse, based on the gravitational pull between the two planets and the sun. The calculations of the movement of the comet could be done after the calculations for Saturn and Jupiter. Whereas the position of the two planets would pertube the orbit of the comet, the effect of the gravitational pull of the comet to the planets could safely be ignored[72, 307].

Clairaut's approach allowed for treating the complex relationship between Saturn, Jupiter and the sun, but in turn it required a monumental amount of calculations to be made to compute the path of the comet for roughly 75 years forward in time since its last observation in 1682. He began the computation during the early summer of 1757, along with two assistants, Joseph Jérôme Lefrançois de Lalande(1732–1807) and Nicole-Reine Lepaute(1723–1788). The comet's path could be computed independently once the intertwined movements of Saturn and Jupiter was dealt with, and the three mathematicians therefore divided the computational work in such a way that de Lalande and Lepaute would handle the first part of the computation, the three-body problem, and Clairaut would use their results to compute the path of the comet as well as checking for any errors. The latter was of great importance since any arithmetic mistakes not caught could compound over time and render their predictions useless. The problem of avoiding errors was at the time considered a great weakness of the then new field of numerical techniques[142]. Critics of scientific research felt certain that the inevitable introduction of small errors would render such methods much too frail to capture the nature of the universe.

Despite the critics, the three toiled away with tedious calculations from morning to evening for almost six months under the increasingly stressful knowledge, that the comet may arrive before they'd finish their work. In November 1757, the three announced their prediction that the comet would reach its perihelion on April 15th 1758. Clairaut was, however, cuatious in his prediction as he knew that small quantities neglected in the approximations would compound over time. He therefore suggested in his announcement that the comet may come as early as 30 days before or as late as 30 days after the computed date[72]. Astronomers saw the first sign of the comet January 1758 and the comet reached it's perihelion on March 13th, just a few days outside of the predicted window of arrival. As the news spread of the discrepancy between the predicted arrival and the actual date, it sparked a vivid discussion amongst astronomers if it was due to errors in the calculations, the approximation or if other things were at play. Despite the apparent success, it did not appease the critics of numerical techniques. The discussion continued for years and notable voices such as Jean-Baptiste le Rond d'Alembert(1717–1783) were amongst those who decried the "spirit of calculation"[307].

Today it is known that the effect of the, at the time unknown, planets Uranus and Neptune were amongst the factors contributing to the discrepancy between the date computed by Clairaut and his assistants and the actual date of arrival. The achievement, however, still stands as probably the first ever large scale parallel computation to seek an approximate solution to a set of differential equations.

For the second anticipated return of Halley's comet in 1835, there were several major attempts to predict the time of the perihelion, all using similar technique to that of Clairaut. Uranus had been discovered since the last passing and it's effect could therefore be included in the computations. On average, the predictions for the 1835 arrival were 16 days off the date of perihelion, half the error of Clairaut's original prediction for the 1758 return. For the 1910 return, interest in the comet had declined substantially and only Andrew Claude de la Cherois Crommelin(1865 –1939) at the Royal Greenwich Observatory made an attempt at predicting its arrival time. Crommelin's novel contribution was to discard the previous approach of successive corrections to ideal elliptical paths and instead use the novel method of "mechanical quadratures", now known as numerical integration[75]. With the help of the Greenwhich observatory computing staff he predicted 17th April 1910 to be the date of return. The comet reached perihelion on the 20th April, just two days and seventeen hours late of the prediction.

At the time of the second and third anticipated return, no one doubted the validity of Newtons theory of gravity nor did anyone doubt the usefulness of numerical techniques. The return of the comets, and their predictions, therefore didn't receive the same attention from the public and the broader scientific community as that of the first anticipated return. In the 152 years that had passed since the first anticipated return, intricate calculations carried out by human computers had found many purposes and was no longer a rare oddity. Large groups of human computers were employed to compute celestial tables for the annually released nautical almanacs used for navigation at sea, and many governments had dedicated bureaus, employing human computers to compile statistical data for use in governance. Some large cooperations such as General Electric even had a dedicated office of human computers, working to assists engineers with solving problems that required a substantial amount of calculations[142]. The arrival of commercially viable mechanical calculators in the mid to late 19th century had further improved the efficiency and capabilities of the human computer but also hinted at the beginning of a new era in computing[306]. By the early 20th century, the human computer began to see competition on certain tasks in the form of electromechanical tabulating machines. The machines were expensive to buy and operate, they needed cumbersome punch cards, and a substantial amount of work was required when machine operation needed to be modified[161]. Despite these shortcomings, the machines turned out to be economically viable alternatives to human computers for certain simple tasks such as accounting and tracking of inventory.

Human computers were likewise reaching their limits in terms of capacity. New problems began to arise in science and engineering that required calculations of a magnitude hardly feasibly with human computers. During the course of World War I, the English physicist Lewis Fry Richardson(1881–1953) had developed a system of partial differential equations to describe how the weather change over time, using just seven basic atmosphere variables: air density, pressure, humidity, temperature and velocity in 3 dimensions[199]. Richardson speculated that if one could solve the system of partial differential equations trough numerical integration on a global longitude-latitude grid, it would be possible to predict the weather based on first principles rather than on extrapolation and statistics as was done at that time. Using 2000 points, he estimated that one would need 64.000 human computers working in parallel to track the weather in real time and more to do actual forecasts. It would be more than 30 years, from the time Richardson first began his work on numerical weather forecasting, until it first saw practical use. In the mean time, another problem had far greater priority amongst decision makers: Calculation of ballistic trajectories.

During the course of the first and second world war, a new problem had presented itself to army generals. Developments in weapons technology in the late 19th and early 20th century saw the introduction of new types of canons and artillery with a range and use for which old methods of computing firing tables were insufficient. New long range artillery could propel shells so high in the air, over such great distances, that even changes in air density and the rotation of the earth had to be accounted for to accurately predict the point of impact[134]. Accurate firing tables for the new anti aircraft artillery turned out to be particularly difficult to create as old methods completely failed for high-angle firing. Fuses on the projectiles had to be timed to explode in the air in the vicinity of a target aircraft, so the complete trajectory of a given shell was needed to create the appropriate tables and not just endpoints and time of flight.

The problem could be modeled well by a differential equation accounting for the various forces that acted on the shell. To find an approximate solution to the equations trough calculations, the newly developed methods of numerical integration would find use. While solving the problem was not nearly as complex and time consuming as the three-body problem, first tackled by Clairaut and his assistants more than 150 years earlier, it still took a trained human computer with a mechanical calculator one or two full days of work to compute a single trajectory[134, 142]. The pressing issue was that for each type of ammunition and each type of artillery, trajectories would have to be calculated for many different conditions such as angle of elevation, crosswind effect, motion of the barrel, and local firing conditions. The large parameter space made the work of such magnitude that it was difficult for the army corps of human computers to deliver in due time[136].

Artillery played a major role in the early days of industrial warfare. The majority of combat deaths during the Napoleonic wars and both World Wars were due to artillary bombardments[162]. In connection with the ongoing war effort of the United States during World War II, cost effective and accurate use of artillery was deemed of great importance and the funding environment for developing new computing machines to aid the war effort had therefore become very generous.

There had been many early attempts at creating computing machines to do the work of the human computer. Charles Babbage(1791–1871) is often credited with designing the first Turing complete programmable computer, the *Analytical Engine*, though it was of such mechanical complexity that it was never built. Babbage had earlier been working on building his *Difference Engine* funded by the British government[150]. The machine was supposed to make the calculation of astronomical tables for aid in navigation more economical, but it too was of such complexity that he was unable to finish its construction before the government grew impatient and abandoned the project.

The first major success for electro-mechanical computers came as part of the efforts during World War II. The US, UK, Germany and Japan all funded projects to build large computing machines to tackle problems that were difficult or even impossible for human computers to solve. In addition to computing ballistic trajectories, the machines were to be used for other military applications also such as code breaking and the design of nuclear weapons.

The most famous of early machine computers was the *Electronic Numerical Integrator and Computer* (ENIAC) completed in 1945 at the University of Pennsylvania for the Research and Development Command of the US Army Ordnance Corps. Unlike other machines at the time, it wasn't kept secret but put forward to the public on display to be celebrated by the press as a gigantic brain. ENIAC could compute a complete ballistic trajectory in just 30 seconds, a task that would otherwise take a human computer 20 hours. When construction had started in 1943, it was intended to be used mainly for computing ballistic trajectories, but by the time it was completed and ready for operation at the end of 1945, the war had ended and the calculation of trajectories was no longer an urgent matter[149]. Its first task instead became to do calculations for the design of thermonuclear weapons. Aside from the secret *Colossus* computers build in the UK for cryptographic purposes, the impact of the new machines on the war effort was very limited due to their late arrival. The government spending spree did, however, have other positive benefits, in particular for the scientists who were fortunate enough to use the machines that the army had paid to build and develop. In 1950, the ENIAC machine was used for the first crude numerical weather forecast as envisioned by Richardson more than 30 years earlier whilst working as an ambulance driver during World War I[281, 142].

Although the large new computing machines built during the late 1940s and 1950s could theoretically deliver the computational capacity of several thousand humans, they did not displace the use of human computers right away. The machines had many problems of their own. They were expensive to buy, they typically relied on vacuum tubes that consumed a large amount of electricity, and they would often break. Maintenance and operation was cumbersome and difficult. Due to the limitations of early day machine computers, the widespread use of human computers would continue for another decade in tandem with the emerging machines. Up until the beginning of the Mercury program, aiming to carry humans into space, NASA was still relying on human computers for some mission critical calculations[138].

The invention of the semiconductor transistor, for which the Nobel prize in physics was awarded in 1956, marked the beginning of a new age. Machines using integrated circuits became commercially available in IBM computers in the mid 60s and Intel released the world's first microprocessor in 1971[197]. During the next several decades the world would see an exponential increase in computational power that would come to revolutionize not only computational science and engineering but society as a whole.

2 Frontiers of Computational Science and Engineering

Modern day computational science has come a long way since the french astronomer A.C. Clairaut, along with his two assistants J. Lalande and N.R. Lepaute, worked on tedious calculations of formidable scale for almost six months to compute the orbit of Halley's comet and predict the time of its return[72]. If, at the time, A.C. Clairaut and his two assistants had access to the computing machines available to modern day scientists, solving the same problem would have taken less than a second.

Today, an off-the-shelf laptop will easily compute at a rate of 10^{10} floating point operations per second (flop/s) and the compute performance of a powerful workstation may well be measured in teraflops (10^{12}flop/s). The most powerful computing machines available to modern day scientists are constructed by connecting thousands of high-power computers in high-speed low-latency networks. The compute performance of the most capable of these machines is now measured in petaflops (10^{15}flop/s). In comparison, the earliest electronic general-purpose Turing-complete supercomputer, the ENIAC (Electronic Numerical Integrator and Computer), was able to perform anywhere between 40 and up to 5000 operations per second, depending on the type of operation[137, 254].

This colossal increase in computational power over the past century has gone hand in hand with advances in mathematical modeling and computational mathematics so that it is now possible to solve problems and simulate phenomenons previously considered impossible. Today, computational and simulation science is regarded as a third pillar of the method of scientific discovery. Computers have become a vital tool, complementing theory and physical experiments, allowing scientists to explore phenomena that are otherwise infeasible to investigate. As for the scientists, computer simulations and data science have become an ubiquitous and indispensable tool for engineers in everything from designing airplanes to discovering new medicines and materials [175, 57, 309].

Since 1993, the TOP500 project has published a list of the worlds fastest supercomputers twice a year[209]. Systems are ranked according to their performance on the Linpack benchmark which solves a dense system of linear equations[94]. The list has come to serve as the defining yardstick for supercomputing performance, and the data compiled and collected over the years is often used both for describing the history of supercomputing and for predicting future developments [208, 292]. As of November 2017, 181 systems are registered as being able to deliver one petaflops or more on the Linpack benchmark. The last entry on the list, the 500th computer, achieved 0.549 petaflops of sustained computation in the Linpack benchmark.

The peta-scale machines are modern-day miracles, but 10^{15} floating point operations per second comes at the cost of complexity of use. A lot of complexity. To write a scientific code that run efficiently on the entire machine is no simple task. The machines on the TOP500 list typical have several thousand computing nodes with separate memory spaces for which the nodes must globally coordinate, synchronize, and exchange messages when solving a problem that is not embarrassingly parallel. In addition, the compute nodes themselves are complicated machines with increasingly convoluted memory hierarchies in a shared memory space of many cores, on potentially many sockets with non-uniform bandwidth access and latency. To make matters worse, supercomputers are increasingly pushing the limits of compute performance by adding compute-accelerators, or co-processors, with specialized processing capabilities such as GPUs[227, 44], DSPs[268, 168], FPGAs[245] or a combination hereof[299] to handle particular tasks. This addition of hardware, specialized for certain compute patterns, is often denoted as “heterogeneous computing”. Heterogeneous computing is a major development as it leads to faster and more energy-efficient computers[195, 214]. However, it also add yet another level of complexity for the users of the machines as they now have to handle many nodes, each with potentially many different instruction-set architectures with each their own complexities. Due to the complexities involved in going from science to hardware and back again, scientific code running on the full size of peta-scale clusters is now often written by large teams of both HPC programmers and domain scientists.

In 1985, a new annual prize grew out of a SIAM meeting when a group was discussing the idea of having a prize to recognize the speedup of real applications on real parallel machines so to track how and if advancements in compute power was leading to real-world progress in science and engineering. Alan Karp, at the time a staff scientist at IBM, had observed that even though there were plenty of talks about building systems with 1,000 and even 10,000 processors, nobody had demonstrated that reasonable speedup could be obtained on much smaller systems already available. Karp decided to pose a challenge to the community, offering US\$100 to the first person or group to demonstrate a speedup of at least 200 times on a real problem running on a general purpose parallel processor. Gordon Bell, then the founding Assistant Director of NSF’s Directorate for Computer and Information Science and Engineering, thought Karp’s challenge was a good idea, so he decided to make his own annual price, originally set at US$1000, for the best speedup of a real application running on a real machine[27]. The prize has now been awarded annually since 1987 and is today known as the ACM Gordon Bell Prize, given each year at the SC Conference with the winning team sharing US$10,000. Winners are selected according to the following criteria

> “The Gordon Bell Prize is awarded each year to recognize outstanding achievement in high-performance computing. The purpose of the award is to track the progress over time of parallel computing, with particular emphasis on rewarding innovation in applying high-performance computing to applications in science, engineering, and large-scale data analytics. Prizes may be awarded for peak performance or special achievements in scalability and time-to-solution on important science and engineering problems[8].”

Given the amount of publicity and attention the prize has drawn from the community over the years, it is not unreasonable to assert that data on past winners and finalist may be used as a proxy to track the development of state-of-the-art computational science applications running on state-of-the-art supercomputers. Figure 2.1 contains a scatter plot of prize winners and notable finalists over the past decade, showing sustained computational performance as a function of year. In 2007, the winners had demonstrated the first ever micron-scale molecular dynamics simulation, developing the Kelvin-Helmholtz instability. The authors had used advances in fault tolerance, kernel optimization, and highly efficient parallel I/O to squeeze 115 teraflops of effective continuous compute performance from the 131,072 cores of a BlueGene/L machine[133].

In recent years, demonstration of multi-petaflop/s performance on a real-world application has been necessary to claim the prize. Last years winner was a 12-member Chinese team, demonstrating 18.9 petaflop sustained compute performance whilst running a simulation of the 1976 Tangshan earthquake on the worlds largest supercomputer, the Sunway TaihuLight situated at the National Supercomputing Center in Wuxi, China[118]. The simulated physical area spanned 320 km by 312 km, as well as 40 km below the surface resolved to 8m. In total, 450TB of input seismic data was used[116]. Another 2017 finalists, also a Chinese team using the Sunway TaihuLight, ported, redesigned and scaled the Community Atmosphere Model (CAM) to the full system of the Sunway TaihuLight supercomputer[225, 117]. With a global 25km resolution they were able to simulate 3.4 years per day. At 750m global resolution, they measured a sustained performance 3.3 petaflop/s. With the high-resolution global model, they were able to simulate the complete lifecycle of hurricane Katrina and achieved close-to-observation simulation results for both the hurricane track and the intensity, the first such feat.

With the introduction of Petascale machines, the accurate simulation of global weather phenomena[117], large earthquakes[116], blood flow at cell resolution[256], among many other things now possible. In the next chapter we give a brief overview of what potential scientific advances the introduction of Exascale computing machines may lead to. We follow with a discussion on both existing and speculative challenges in bringing applications to these new machines.

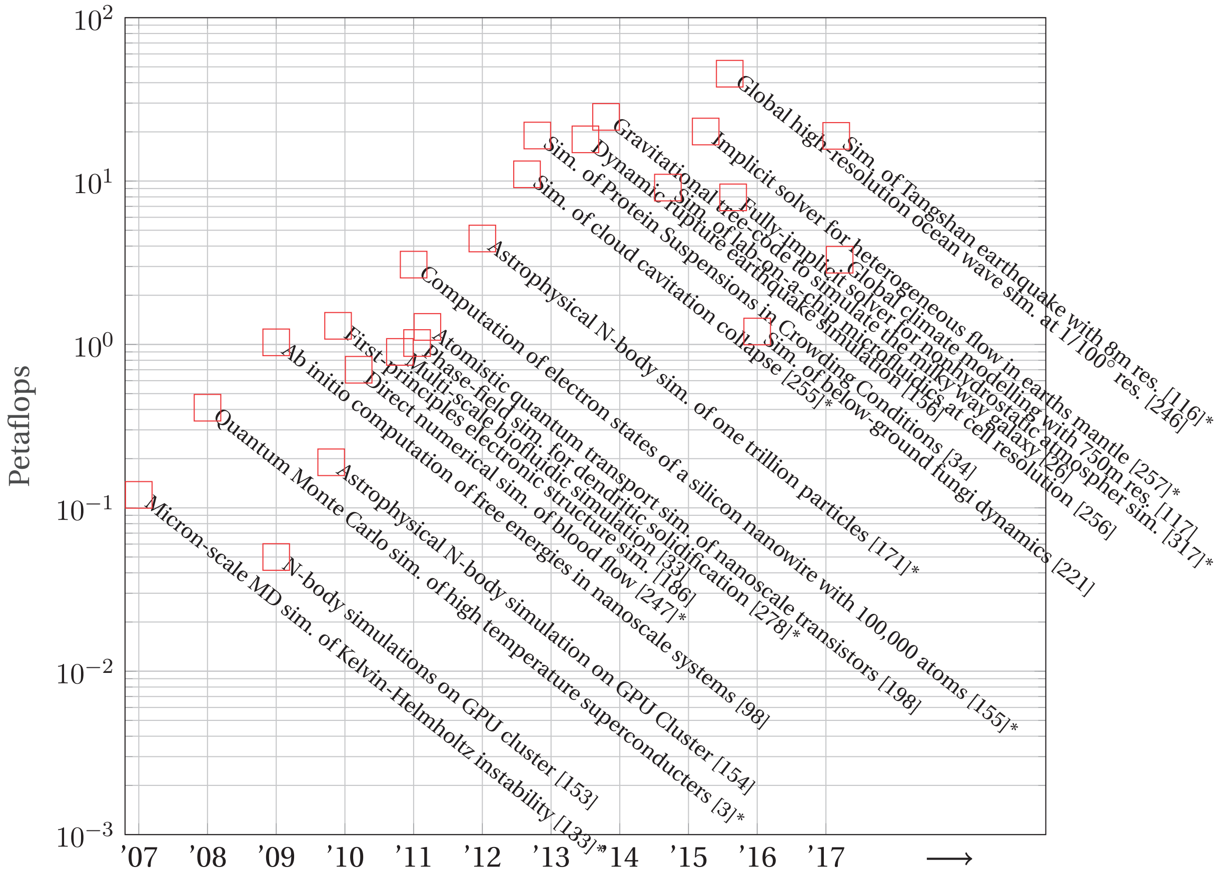


Figure 2.1 – List of major landmark computations in scientific computing on top supercomputers. The list is not meant to be exhaustive, and only results for where a cite-able source, with details on the study, could be found, have been included. The * indicates that an associated paper was awarded the Gordon Bell prize in either the Performance or the Scaling category. Most entries of dates have been moved slightly left or right so to allow for readable labels.

3 Moving to Exascale: Status and Open Challenges

This chapter begins with a brief description on what motivates the global push for Exascale capable supercomputers. The description is followed by an overview of currently known planned Exascale machines and the timeline for their development. Finally an introduction to the challenges, expected in making real world computational science and engineering application scale to the full size of these machines, is given.

3.1 Motivation for Exascale

The worlds first supercomputer to exceed 1 petaflop of compute performance on the Linpack test went online in 2008. It was the IBM built Roadrunner, using PowerXCell 8i processors, a processor variant IBM designed specially for HPC[16]. In 2012, four years later, there was 20 Petascale systems deployed across the world, with many more on the way. Today, all systems on the TOP500 list are rated at 0.6 petaflops or more[209]. The community has come a long way over the course of the past decade in making applications scale to the full size of these new, often heterogeneous, massively parallel machines. The machines have been, and are being, used to solve problems in science and engineering previously beyond the realm of possibility. Petascale computing has become mainstream in high-end supercomputing, and the community is now beginning to look ahead at what new problems could potentially be solved with access to Exascale capable machine. A number of studies on such advances in the applications of computational science and engineering has been published and below follows a non-exhaustive list with excerpts and conclusions from these studies.

Climate Science Climate modeling is an application where Exascale computing has the potential to make significant impact. The goal of computational climate modeling is to estimate the response of the global climate and temperature to increases in greenhouse gases. To simulate how these processes evolve over time requires addressing an intricate multi-physics problem, combining mathematical models for atmosphere, ocean, ice sheets, land surfaces, and biosphere dynamics. Complicated temperature feedback effects of melting ice caps, ocean circulation patterns, and natural carbon stores further complicates the models[183]. Even with available Petascale machines, resolution and fidelity of simulations still leaves much to be desired. Furthermore, many studies are meant for policy makers and therefore must include estimates of uncertainties in predictions[237]. Climate science is a field hungry for flops.

Material Science Development of new materials play a key role in finding solutions to many technical challenges faced by society. In particular, the development of new materials for energy storage and photovoltaics are considered key in moving towards the widespread adoption of renewable energy sources. Petascale machines has allowed for atomistic scale simulation and design. Exascale machines are needed to close the gap from atoms to complex heterogeneous materials since simulation of the latter requires the bridging of processes spanning from nanometers to micrometers and femtoseconds to minutes. As an example, the accurate modeling of lithium-ion batteries requires atomistic modeling at micron length scales over many cycles of charging and discharging to gain insight into modes of failure and degradation. Multi-scale, multi-physics models, connecting transport of ions and electrons, electromechanical interface reactions, heat generation and transfer and structural deformation and stress, have been developed for such simulations, but the numerical solution at scale remains challenging. One issue is the many-body nature of parts of the models. The number of operations for ground-state calculation using density functional theory (DFT) scales cubically with the number of atoms, other models scales even worse. Another complexity is the enormous search space of potential materials to investigate. Current state-of-the-art high fidelity models require peta-scale machines. To evaluate the properties of thousands of candidates, Exascale compute capacity is clearly needed[120, 92].

Fusion Energy The 4th state of matter, plasma and ionized gases, have many important industrial applications, particularly in semiconductor processing and in medical applications. It is also at the center stage of the scientific grand challenge of the realization of fusion energy. Accurate simulation of plasmas is a difficult problem, in particular when it is necessary to also simulate the interaction of burning plasma with solids such as the effect of fusion products on structural materials of a fusion reactor. The simulation of plasma involves solving the Boltzmann equation, coupled to Maxwell's equations for the evolution of magnetic and electric fields, and for many practical problems, one must also couple to other equations for atomic, molecular, and nuclear processes. While major advances has been made over the past decade with Petascale computing, Exascale computers are considered critical as a next step to capture the extreme range of mutually interacting temporal and spatial scales and to enable the multiphysics modelling necessary to address many essential questions [59].

Medicine Supercomputing has become an essential component in medical and healthcare innovation. Due to the scale and complexity of developing new drugs and treatments, doing so has traditionally been a very costly and time-consuming human-driven affair. Medical trial and error laboratory-based research typically involves long time lines and supercomputers are therefore increasingly being relied on to accelerate discovery. In the field of cancer research, projects are underway to compile all known data on how cancer functions and how drugs reacts and behaves using Exascale machines. The projects involve deep learning and novel data acquisition techniques to be able to create prognosis and treatment plans, designed specifically for the individual patient, based on all available knowlegde[251, 164, 277].

Brain simulation The human cortex consists of roughly 10^{10} neurons, each neuron being connected trough synapses with roughly 10^4 other neurons for an estimated total 10^{14} synapses. Currently, it is possible to simulate up to about one percent of the neurons in the human brain with all their connections using Petascale machines. It is estimated that with an Exascale machine it would be possible to simulate the entire human brain at the neuron level[205, 179].

Turbulent Flows Simulation of turbulent flows has numerous applications in combustion, atmosphere, wind, fusion and nuclear sciences. It's been estimated that access to Exascale compute resources will enable a higher degree of system level simulations of turbulent flows. In wind energy, one could integrate the simulation of individual wind-turbines with the simulation of wind-farms with hundreds of turbines, in combustion and vehicle design it would allow for connecting simulation of in-cylinder combustion processes with models for gas exchange processes, turbocharging and the drivetrain. These tightly couples models would facilitate a significant improvement in overall fidelity and correspondence with real world performance[284]. Computational scientists in turbulent flow also envision using Exascale machines for solving turbulent flow optimization design problems by solving Petascale problems 1000's of times. This is expected to be particularly challenging as gradient based optimization algorithms generally do model evaluation and improvements sequentially, in effect implying that the Petascale sized problem must be solved efficiently on the Exascale machine which is almost certain to bring about difficulties with sufficient strong scaling. The concept of strong scaling and related general issues with scaling numerical algorithms to large machines is discussed in Section 3.3.

Multi-Scale Modeling Many applications have important features at multiple scales of time and/or space that must all be captured to correctly model a system. Multiscale modeling thus finds a wealth of applications across engineering, physics, chemistry, biology, meteorology and even operations research. The need for large multiscale simulations is a primary motivational driver for Exascale. Recent studies have demonstrated how multiscale computing may be particularly suited for Exascale HPC systems by improving load balancing and by introducing energy aware computing [182, 2].

Uncertainty Quantification In computational science, simulations often need input parameters and/or real world data that is subject to some form of uncertainty. In many fields, there is an increasing demand for methods to quantify how uncertainties in input parameters lead to uncertainty in the outcome of simulations and forecasts. In the growing field of Uncertainty Quantification, researchers typically split the approaches into intrusive and non-intrusive methods. The non-intrusive methods, e.g. various types of Monte Carlo and stochastic collocation schemes, reuse existing tools and generate a series of deterministic solutions to approximate statistical moments of an output. These approaches are popular because they do not require re-implementation of the original solver. The downside is that their weak approximation capabilities may lead to inaccurate results. The intrusive techniques simultaneously discretize stochastic and physical space, which leads to much stronger approximation properties and therefore an overall smaller run-time if sufficiently small errors on the uncertainty is required. Either approach is, however, still extremely computationally expensive for complex problems even for todays Petascale machines and hence a strong motivation for Exascale computers[240, 157].

Many fields of science and engineering stand to benefit from increased compute capacity. In addition to the direct effects of enabling new landmarks runs and opening up new frontiners in research, observers expect that the many technologies, developed for Exascale computers, will subsequently move into mainstream HPC and industry [184, 233, 132, 163]. In the next section a brief overview of planed Exascale compute facilities worldwide is given.

3.2 Planned Exascale Supercomputing Facilities

Several countries around the world have presented plans to build machines with Exascale compute capabilities within the next 5 years[180]. Historically, the US has been the nation spearheading this, supplying most of the entries on the TOP500 list of the worlds most powerful supercomputers. The US Department of Energy plans on spending USD 1.8 billion ($\sim$EUR 1.5 billion) on building two Exascale supercomputers at US National Laboratories in the 2021-2023 time-frame[300]. The first machine, named Aurora, is currently under development by Intel and Cray in corporation with Argonne National Laboratory and is scheduled to go online in 2021. Few details on the design of the system have emerged as all involved parties are subject to some form of nondisclosure agreement. Thus far, the only information released on what to expect of the new machine is from a recent Call for Proposals by Argonne Leadership Computing Facility looking to select 10 new projects to be part of the Aurora Early Science Program[6]. In their Guidance About Aurora for Proposal Authors, the following points are highlighted about the hardware to be expected

- “The compute performance of the nodes will rise in a manner similar to the memory bandwidth so the ratio of memory bandwidth to compute performance will not be significantly different than systems were a few years ago. A bit better in fact than they have been recently.”
- “The memory capacity will not grow as fast as the compute performance so getting more performance through concurrency from the same capacity will be a key strategy to exploit the future architectures. While this capacity is not growing fast compared to current machines it will have the characteristic that the memory will all be high performance alleviating some of the concerns of managing multiple levels of memory and data movement explicitly.”
- “The memory in a node will be coherent and all compute will be first class citizens and will have equal access to all resources, memory and fabric etc.”
- “The fabric bandwidth will be increasing similar to the compute performance for local communication patterns although global communication bandwidth will likely not increase as fast as compute performance.”

The machine is thus expected to be not entirely dissimilar from current large Petascale machines, but with increased compute performance and energy effciency on the node level and increased bandwidth between nodes. Also, it is noted that as has been the case for a while, global communication bandwidth will not increase as fast as compute performance. The second machine scheduled for 2022-2023 is to be based on R&D in the ongoing Department of Energy Exascale Computiong Project (ECP). The machine might be based on novel memory-driven computing technology, currently being developed by Hewlett Packard Enterprise on a grant by the DoE ECP[158, 93].

The EU has it’s own Exascale Computing Project with a time-line similar to that of the US. On January 11th 2018, the EU launched a new EUR 1 billion project to build an Exascale capable supercomputer by 2023. The project will receive EUR 486 million from the European Commission’s Horizon 2020 budget, an equivalent amount will be supplied by the 13 member states that signed up for the project. A new organization, the EuroHPC joint undertaking, will be set up in 2019 for the project. The declared goal is for Europe to have two "world class" supercomputers, each capable of at least 100 petaflops by 2020. This is in preparation for the final goal of building an Exascale system by 2022-2023[103, 104]. Since the project is still in its infancy, no details exist on the planed machines, though it is noted that the project is “to support the development of European supercomputing technology, including the first generation of European low-power microprocessor technology”. This statement likely points to Asos/Bull as a contributor as it is currently the only European company with expertise in building ultra large systems. France appears to be the sole country in Europe with a roadmap to develop and deliver an Exascale supercomputer. The french Commissariat à l’Énergie Atomique et aux Énergies Alternatives (CEA) has contracted Atos to develop a machine capable of scaling to Exascale size by 2021. In addition the agency has signed an agreement with Japanese RIKEN to corporate on the development of an ARM based Exascale HPC ecosystem[223].

Among the nations planning to build Exascale supercomputing facilities, Japan was the first to announce concrete plans. In 2014, plans was put forward for a Post-K computer capable of Exascale computation to be build by Fujitsu and the RIKEN Advanced Institute for Computational Science. The Japanese government has budgeted JPY 110 billion ($\sim$EUR 850 million) towards the project that was named FLAGSHIP2020. The aim was for the computer to be deployed by 2019 and go into production in 2020. The project has, however, been delayed, most recent reports suggest by 1-2 years. For the Post-K computer, Fujitsu has announced that they will move away from the SPARC64 architecture used by the company for previous generations of high-end supercomputers, and instead move to a new vector-enhanced ARM architecture[170].

In recent years, China has emerged on the scene of international high-performance computing. A decade ago, June 2008, China had only 12 supercomputers among the worlds TOP500 most powerful machines and a combined performance share of only 1.2%. At that time, machines in the U.S. accounted for 61.6% of the installed supercomputer compute capacity worldwide. A decade later, China has come to rival the US in terms of installed capacity. On the November 2017 update of the TOP500 list, the Chinese TaihuLight at the Chinese National Supercomputing Center in Wuxi, build by Sunway was the worlds most powerful supercomputer. Among the 500 most powerful supercomputers, 202 systems are installed in China accounting for 35.4% of the supercomputer compute capacity installed worldwide[209]. China’s recent efforts on pre-Exascale systems has been somewhat hampered by technology export bans instituted by the US in early 2015. The Chinese are, however, showing no signs of slowing down their ambitions, and are consequently investing heavily in homegrown processor and interconnects hardware. The Sunway built TaihuLight was made entirely using Chinese-designed interconnect hardware and many-core processors. Under the country’s 13th 5-Year Plan released in 2015, China committed to launching its first Exascale supercomputer by the end of 2020, or possible 2021. Reports of their progress is difficult to come by though, as little information is released and when it is, it is usually trough official Chinese media outlets with few details. It appears that there are multiple ongoing projects at the six National Supercomputing Centers of China located in Guangzhou, Changsha, Jinan, Shenzhen, Tianjin and Wuxi. The Guangzhou and Wuxi centers are already home to the two most powerful supercomputers in the world.

According to Chinese national news media, another two Exascale prototype systems are due to be completed in 2018 at the Jinan and Tianjin centers[68, 316]. In addition, a fully capable Exascale machine is to be build at the center in Shenzhen. The Shenzhen machine is planned to become operational in 2020 at a cost of CNY 3 billion (∼EUR 400 million) according to China Daily[69]. Some industry observers expect China or the US to become the first country to demonstrate an Exascale supercomputer[272]. Setbacks and delays are, however, not uncommon when building novel high-end systems and the announced Exascale system time-lines of both the US, EU, China, and Japan overlaps to some degree. It remains to be seen by who and how the worlds first Exascale capable machine will be build and operated. In the next section, an overview of the future challenges of making numerical algorithms scale to the full size of these emerging machines will be presented.

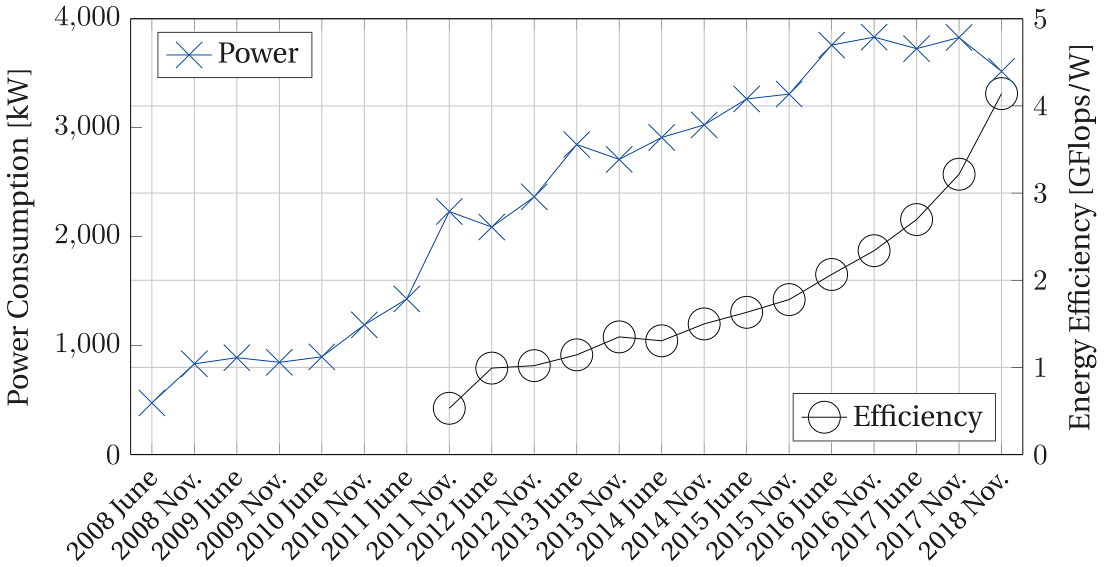


Figure 3.1 – The average energy consumption and average compute efficiency of the 50 most powerful supercomputers in the world as reported on the TOP500 list[209]. Tracking power usage was only introduced in 2008 and compute efficiency in 2011 and data was not available for every machine on the list for all years. The data points are therefore computed as averages of the TOP50 machines for which data was available in a given year.

3.3 Exascale Challenges on Exascale Machines

Challenges related to Exascale computing are not only related to building machines with a theoretical compute capacity in excess of an exaflop whilst insuring that all nodes can communicate with a sufficiently high bandwidth and low latency to allow for Exascale capability on real problems. The user aspect of writing code and formulating numerical algorithms that may scale to the full size of the machine efficiently whilst

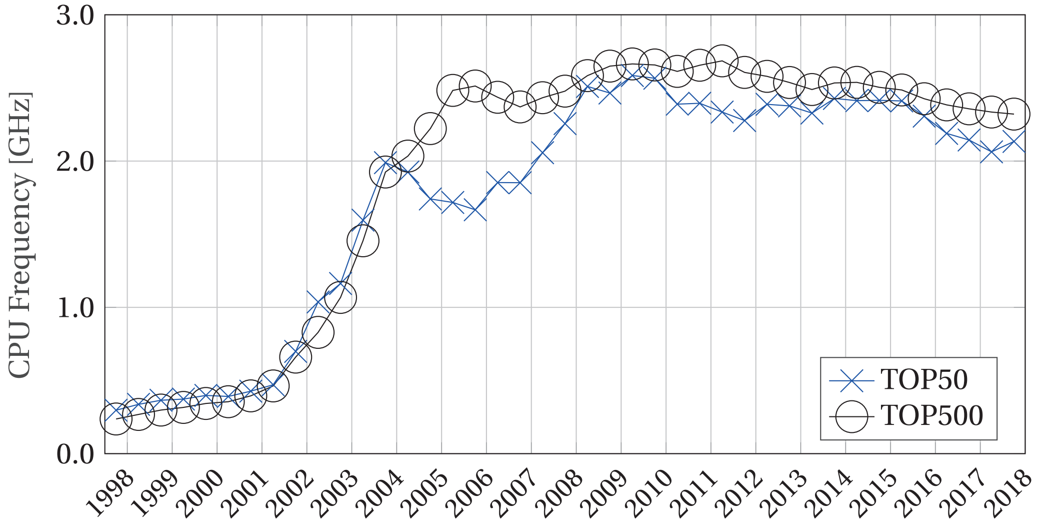


Figure 3.2 – Average CPU frequency of the 50 most powerful machines on the TOP500 list and of all machines on the list. The frequency is averaged over the main processor, i.e, not including coprocessors. Figure compiled using freely available data from TOP500 lists dating back to 1998 [209].

producing reliable trustworty output in itself presents a tremendous task. In a recent report, titled *Applied Mathematics Research for Exascale Computing*, released by the U.S. Department of Energy, a number of expected challenges in making numerical algorithms scale efficiently to Exascale machines is outlined and described[92]. The five main concerns are, power consumption, extreme concurrency, limited per-core memory, data locality, and resilience. In the list below a brief description of the issues foreseen is given.

Power The currently most power efficient supercomputer on the TOP500 list is the Japanese built 0.8 petaflop/s *Shoubu system B* by PEZY Computing. Each node in the system consists of an Intel Xeon host CPU and 8 PEZY-SCx co-processors which act as accelerators. The system is able to deliver 17Gflop/s per watt[209]. If one was to extrapolate this machine to Exascale, it would consume at least 60MWe. Possibly more due to the added energy consumption of an extended network fabric to support the larger machine. In comparison, 60MWe is enough to power a small city with ∼100.000 residents and the worlds currently most powerful machine consumes 15MW of electricity at 6Gflop/s per watt also using purpose build processors and accelerators. The typical commodity hardware type machine on the TOP500 list is able to achieve 1-2Gflop/s per watt, corresponding to a power-consumption of around 600MWe at Exascale which is equivalent to the output of a medium sized nuclear power plant. Figure 3.1 contains a plot of the power consumption and compute power efficiency over the past decades of world-class systems. The power consumption has been steadily increasing since the inception of the list in ’93 to the point that the life-time cost of electricity of running the machines is now comparable to the cost of the machines themselves. This development is driving a change in architecture for high-end HPC and in power-aware computing in general[172]. It is speculated that energy used by a job/simulation may compliment, or even replace, CPU time as the cost metric for supercomputer use. In big machines, moving data between nodes is as costly energy-wise as working on the data, suggesting that numerical algorithms may thus need to become power-aware in the future.

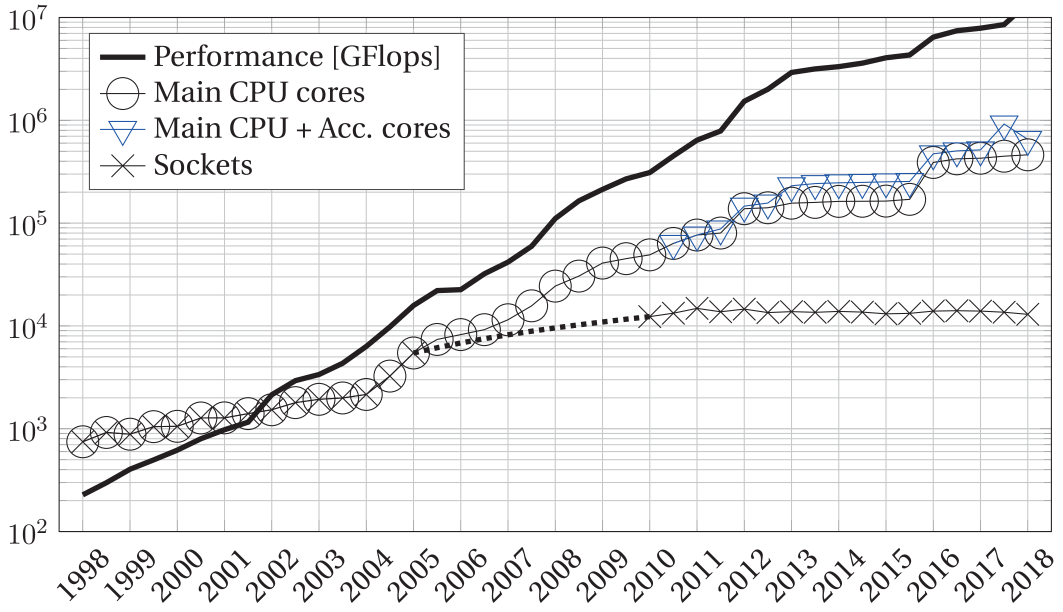
Concurrency Processor clock speeds has been stagnant for more than a decade now due to power density limitations, see Figure 3.2 which contains a plot of the average frequency as a function of time since 1998 of the worlds most powerful supercomputers. Fundamentally, a transistor is a little toggle that needs to accumulate some charge to switch. The time it takes to accumulate the charge is proportional to the current which in turn is proportional to the voltage applied. The maximum speed at which the transistor may operate is thus proportional to the voltage. Unfortunately, power dissipation in a transistor is proportional to the voltage squared times the frequency. In short, power dissipation is therefore proportional to frequency cubed. Processor frequency stopped increasing at the end of the previous decade because the industry hit practical limits in terms of energy cost and capacity to physically remove the heat that the tiny chips generate. Frequency stopped scaling, but performance improvements have continued due to technical advances in integrated circuit fabrication. The advances has enabled smaller transistors thus allowing for more cores with larger caches to fit on the chips. Since 2008, performance has increased mostly due to more cores per socket per node, adding specialized accelerators to the nodes, and adding more nodes to the machines. This trend with overall more cores is expected to continue. The reason is in part due to the cubic power law of frequency to power dissipation scaling, suggesting that one way of creating more energy efficiency machines is by increasing the number of cores and decreasing the frequency at which they operate. An extreme example of this is the Japanese 19PFlops *Gyoukou*, completed in 2017. It achieves an ultra high power efficiency of more than 16GFlop/s per watt through extreme concurrency by the use of co-processors, specifically built for HPC, with a combined 19.8 million lightweight cores operating at just 700Mhz. There are indiciations that increasing core-count and decreasing frequency is a general trend, see Figure 3.2. In 2010, the average operating frequency of supercomputer main processors was around 2.6Ghz, now it is closer to 2Ghz and there’s a clear downwards trend among the TOP50 machines. Another example of extreme concurrency is the planned U.S. DoE Aurora Exascale machine which is expected to have 50.000 nodes[6]. Even on current Petascale machines with ~ 5000 nodes and combined 100.000+ cores, making numerical algorithms scale to the full size of the machines is very challenging. For many problems, synchronization operations and communication between nodes end up limiting how many nodes can effectively work on the same problem long before the machine itself runs out of nodes to use[92]. New algorithms will need to be developed to identify and leverage concurrency in a way that reduce communication and synchronization between separate memory spaces.

Data Locality The bandwidth and latency of the interconnect fabric has for a long time been unable to keep up with the increasing node-level compute capacity, and the trend is expected to continue for Exascale machines[6]. Whereas the energy, used per floating point operation, has been decreasing and is continuing to do so, the energy used for moving data between nodes is not decreasing at the same rate, hence exposing an increasing discrepancy between energy cost of computing and energy cost of moving data between nodes. On the node-level, memory bandwidth has also had problems keeping up with faster chips. Future systems may need to mitigate these issues, in part, by adding new memory technologies and hierarchies such as nonvolatile memory, scratchpad memory and/or deeper cache hierarchies. This in turn means that algorithms will need to be more aware of data locality and seek to minimize data movement across levels[92]. An example of how important the usage of the memory-subsystem is for compute throughput is the discrepancy between the performance numbers reported on the Linpack HPC benchmark (HPL) and achievable performance on real world applications. The HPL benchmark used since 1993 can be misleading in terms of evaluating the capability of machine to solve real world problems. HPL measures the time to solve a dense n by n system of linear equations. Experience has shown that unless an application consists almost exclusively of a lot of matrix factorization and multiplication of dense floating point matrices, it is unlikely to achieve anywhere near the same compute throughput. Many applications of today use a sparse data structure with random access patterns. In addition the applications often need a substantial amount of global synchronization and communication between separate memory spaces to the extent that they are effectively *memory bound* rather than *compute bound*. For that reason, a new HPC becnhmark has been developed, the High Performance Conjugate Gradient (HPCG) benchmark[89]. HPCG uses a preconditioned conjugate gradient algorithm with lots of collective operations and sparse data structures that stress the memory subsystem both locally and globally[91]. The currently highest compute troughput measured on the HPCG benchmark is 603TFlop/s, achieved by the K Computer at RIKEN Advanced Institute for Computational Science[188], e.g., in HPCG, the petaflop barrier has yet to be broken. A recent study showed just how much the HPCG benchmark stresses the memory subsystem[203]. The researchers demonstrated that knowing just two metrics of the machine being tested: “The effective bandwidth between the main memory and the CPU” and “the highest occurring network latency between two compute units”, one can estimate the HPCG performance quite accurately. The theoretical number of floating point operations that the machine can deliver is therefore of limited relevance when predicting the performance on the HPCG benchmark.

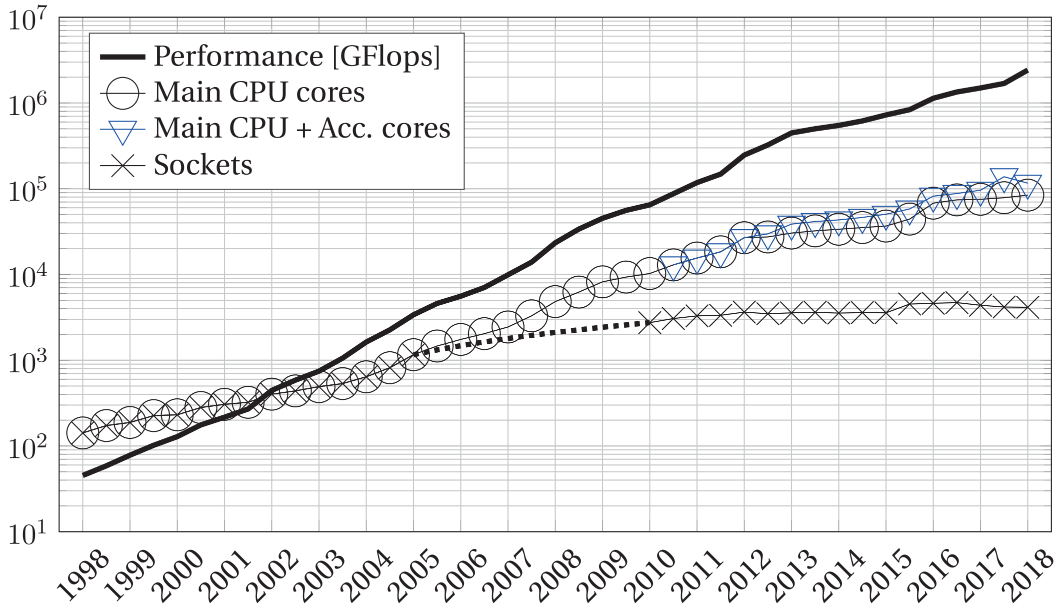
Memory The bandwidth and latency of the memory subsystem and network fabric is not the only growing issue related to memory. Even though the total distributed memory on supercomputers are increasing, the per-core memory is decreasing and this trend is expected to continue[92, 181]. Many current algorithms may thus find themselves to be memory-constrained and will need to be redesigned.

Resilience Due to the sheer number of components, hardware failures are expected to become more common on Exascale machines. Petascale machines are already prone to failures. Even though the nodes themselves may have a mean time between failure (MTBF) of several years, with thousands of nodes in a cluster, the MTBF for a failure occurring on any single node on the entire cluster is suddenly in the order of hours or days at best[51]. This becomes problematic when attempting to run simulations that span an entire cluster as a failure on a single node leads to failure of the entire run. The most common approach used today to mitigate the impact of failures is checkpoint-restart. Applications, meant to scale on big machines, are written in such a way that they periodically dump their memory content to the parallel-file-system. In the event of a failure, the domain scientists running the simulation may then simply restart from the most recent checkpoint[269, 206]. Checkpoint-restart comes at the cost of computational efficiency. The I/O bandwidth of parallel-file-systems continues to not be increasing at the same rate as compute performance and memory capacity. Creating periodic checkpoints often consume a substantial portion of the run time in Petascale applications, wasting valuable compute time waiting for checkpoints to be written. If this trend continues on Exascale machines, it is projected that some applications may be unable to progress as the time to write, and read, a checkpoint to the parallel-file-system will be longer than the mean time between failure. Another potential challenge is mitigating the speculative impact of silent data corruptions (SDC). SDCs may arise as spurious bit flips due to cosmic rays and then propagate throughout the simulation creating erronous results. It is possible that the solution could be corrupted in such a way that it is not immediately obvious to the user that a data corruption has taken place. This in turn raises questions on the trustworthiness of simulations on large machines. In short, the larger the machine, the larger the probability of SDCs[73, 53].

In addition to the many challenges of adapting old, and developing new, numerical algorithms to work well on the emerging machines it has been highlighted that there is a lack of HPC programming skills. Many research centers are forced to train their own staff as very few university provide adequate advanced training in the topic[264]. It has also been highlighted by prominent researcher that there has been an over investment in hardware compared to software over the past decade, leaving scientific applications unable to fully take advantage of new machines[93].



(a) TOP50



(b) TOP500

Figure 3.3 – Average compute performance, number of cores and sockets of the (a) 50 and (b) 500 most powerful machines on the TOP500 list as a function of time. Main CPU cores indicate the average number of cores of the main architecture, used on each machine. Main CPU + Acc. cores indicate the average total number of cores, i.e. including any coprocessor. The dashed line on the socket-numbers between 2005 and 2010 indicates estimates as seperate tracking of cores per socket was not included in the data until 2010 even though multi-core main processors appeared as early as 2005. The two figures were compiled using freely available data from TOP500 lists dating back to 1998 [209].

4 Thesis Content and Contributions

In this chapter a few important general concepts, related to HPC and used throughout the thesis, is presented in the first section. In the second section a brief overview of the content and contributions in each chapter is given.

4.1 Essential Concepts

When evaluating how well the parallel implementation of a numerical algorithm works when running on a supercomputer a few core concepts are widely used. The most essential of these being parallel speedup S_p, defined as the ratio of the elapsed real time of the fastest sequential implementation to that of the fastest parallel solution which is usually written as

$$S_p\left(N\right) = \frac{T_{seq}}{T\left(N\right)}, \tag{4.1}$$

where N is the number of compute resources used in parallel. Another closely related metric is the parallel efficiency typically written as

$$E_p\left(N\right) = \frac{S_p\left(N\right)}{N}, \tag{4.2}$$

whose value lie in the range from 0 to 1. Practitioners are typically interested in how parallel speedup and efficiency change with respect to the number N of compute resources used. A common test to evaluate the performance of an application is to solve a fixed-sized problem several times using different N to see how the parallel speedup and efficiency change as the number of compute resources increases. This type of test is typically referred to as *strong scaling*[141]. Ideal scaling is when the parallel efficiency remains at 100% for all N. If the parallel efficiency at some point become larger than 100%, this is referred to as super-linear scaling. Super-linear scaling can happen for a number of reasons, but is most often related to the availability of more registers and cache when dividing the problem in a distributed memory parallel computing setting[252]. In practice, ideal or super-linear scaling is rare. In most cases overhead, related to communication, synchronization and the limitation of inherently sequential code parts, will result in the parallel efficiency decreasing as the number of compute resources increases. At some point the rate of decrease in parallel efficiency becomes larger than the gain from adding more resources, and the parallel speedup becomes stagnant or may even begin to decrease. Such a peak in parallel speedup is usually referred to as the scaling limit of a given code on a given problem. For a program where only some fixed fraction R may be performed in parallel, this in itself pose an upper limit to obtainable parallel speedup given by

$$S_p\left(N\right) \leq \frac{1}{\left(1-R\right)+\frac{R}{N}} \tag{4.3}$$

The above relation between N, R and $S_p\left(N\right)$ is known as Amdahl's law[5]. In simple terms, if 1% of the program is inherently sequential, then it is impossible to achieve more than 100x speedup, regardless of the extend of the compute resources used. In the early days of parallel computing there was a real concern that this would forever limit the extent to which parallel computing could be used. The first Gordon Bell prize was given for showing that Amdahl's law may not necessarily be a barrier[27]. It turns out that for many practical problems, the computational complexity of some inherently sequential code parts tends to scale differently than code which may be executed in parallel. For example, when doing a large n-by-n matrix-matrix multiplication, the time to load the matrix may be proportional to n^2 whereas the amount of compute work is proportional to n^3. The fraction of sequential code may thus be decreased simply by increasing the size of the problem. As practitioners with access to supercomputers are often interested in solving larger problems rather than solving the same problems faster, this works in their favor. Another often used test to evaluate the performance of an application is therefore to solve a problem proportional in size to N several times using different N to measure how the parallel speedup and efficiency change as the compute resources increase. This type of scaling test is called *weak scaling*. If the elapsed real time to compute remains constant as N increase, the code is said to achieve ideal scaling. For this type of test, Amdahl's law has little predictive power as it assumes that the problem size stay fixed. Gustafson's law addresses this by assuming that the size of the problem to solve increases proportional to the number of compute resources. The expression can be written as

$$S_p\left(N\right) \leq 1+\left(N-1\right)R, \tag{4.4}$$

where N and R are defined as above[148]. Throughout the thesis the concepts of parallel speedup and efficiency in the context of strong and weak scaling will be used to demonstrate improvements in the scalability of numerical algorithms.

4.2 Outline of Thesis

The thesis consists of four parts. Part I is a broad introduction and overview to the field of Exascale computational science and HPC. In Part II, a number of studies on new methods for more efficient scaling of the solution of PDEs is presented. The third part treats aspects in resilience and fault tolerance for numerical algorithms in Exascale HPC. In the fourth and final part of the thesis a summary of contributions is given along with a perspective on the work and some speculations on the future path of Exascale development. The content of each of the two parts containing the main contributions of the thesis is outlined in the following two sections.

Part II – PinT Integration for Scalable Solution of PDEs

After the introduction in Part I follows the first of two parts on the development of numerical algorithms for Exascale machines. The first of these deal with the challenge of scaling the solution of partial differential equations (PDEs) on massively parallel distributed memory machines.

The numerical approximation to a solution to some system of PDEs is generally found by solving one or more linear systems, arising from a discretization of the continuous system of equations. To solve linear systems of the type, typically arising from the discretization of PDEs, domain decomposition methods are frequently used[86]. A key feature of domain decomposition methods is that the spatial domain is divided into many smaller subdomains on which smaller problems are solved locally, independent of the global solution. Consistency with the global solution is achieved iteratively. Within each iteration, boundary information is exchanged between subdomains and a new local boundary value problem is solved on the subdomain. The fact that all the small subdomain problems may be solved independently once boundary information has been exchanged makes the methods highly suitable for distributed memory machines where communication between nodes is expensive in terms of both time and energy.

As long as the size of the global spatial domain is much larger than the number of nodes, such methods tend to work very well. In the strong scaling limit however, subdomains become so numerous and so small that the communication of boundary information becomes a bottleneck for parallel acceleration. This tend to be particularly problematic when some form of time-integration is needed because classical domain decomposition methods only treat boundary value problems. When time dependent PDEs are to be solved, issues arise even in the weak scaling limit since refining the solution in space typically also require some form of refinement in time for accuracy reasons or to satisfy some stability criteria. Consequently, even with ideal spatial weak scaling, the elapsed real time to solution increases when increasing the number of compute nodes used. The scaling issues for time-dependent PDE problems may potentially be greatly diminished by the introduction of parallelism in the time integration procedure. Investigating this is the topic of part II. A brief outline of each chapter in part II is given in the list below.

Chapter 5 Contains a brief overview of domain decomposition methods and an introduction to parallel-in-time integration methods with an emphasis on Parareal.

Chapter 6 Various numerical experiments on the application of Parareal to convection dominated PDE problems are presented.

Chapter 7 A new adaptive scheduler is introduced for Parareal that greatly improve parallel efficiency when the preconditioner is expensive and convergence slow.

Chapter 8 A new approach for constructing Parallel-in-Time integration schemes is presented along with some numerical experiments.

Part III – Fault Tolerant Algorithms for Exascale Systems

Despite very high mean time between failure (MTBF) on HPC hardware, due to the sheer number of components, hardware does occasionally fail. The failure of hardware components is already posing challenges for efficient scaling of algorithms to the full size of current day Petascale machines as outlined in Section 3.3. These challenges are expected to be further exacerbated on Exascale machines[51, 53]. The theme of part III is to explore ways of mitigating these issues in their various forms, in particular those that manifest themselves as either complete notified failure of one or more nodes, or as silent data corruptions, unknown to the program or the user. A brief outline of each chapter in part III is given in the list below.

Chapter 9 A brief introduction to resilience and fault tolerance.

Chapter 10 A new modified Parareal variant, fault tolerant towards complete node failures and to silent data corruptions, is introduced.

Chapter 11 A new C++/MPI library for fault-tolerance using checkpoints and automatic rollback is presented. The library, called Llama, uses multi-level layered group local in-memory checksums to protect distributed arrays.

Chapter 12 Checksum encoding schemes are widely used to protect data from hardware failures. Data protected trough the creation of checksum parity code is, however, only recoverable if the number of lost data and/or code blocks is smaller than the number of checksums parity codes created. In this chapter a preliminary study on partial information recovery in incomplete checksums is presented.

Part II

Parallel-in-Time Integration for the Scalable Solution of PDEs

5 Introduction to Parallel-in-Time Integration Techniques

In High Performance Computing, one typically distinguish between methods that are intrinsically parallel, also called embarrassingly parallel, and those that are not. With intrinsically parallel algorithms, nodes are able to independently work on completely decoupled parts of a given problem, common examples are Monte Carlo type methods and parameter sweeps. Conversely, some algorithms are very difficult to solve efficiently on highly parallel machines. It may be that there are inheritable sequential components to the algorithm that are difficult, or even impossible, to divide amongst many processors; or it may be that when the problem is divided, frequent communication between processors is required during the solution procedure. Many famous algorithms are notoriously hard to solve efficiently on parallel machines, Dijkstra's algorithm[176] and depth-first search[1] being two famous examples that leave little room for parallel acceleration. Another interesting example is that of n-body simulation. If one was to compute all pairwise interactions, the algorithm would be of $O\left(n^2\right)$ complexity, and easy to implement on a parallel machine. Alternatively, one could use a fast summation techniques such as the Barnes–Hut tree, this algorithm is of $O\left(n \log\left(n\right)\right)$ complexity, but highly non-trivial to solve efficiently on a parallel machine[308].

Algorithms for the numerical solution of Partial Differential Equations(PDEs) are typically neither embarrassingly parallel, nor are they inherently sequential with no opportunity for parallel acceleration. Domain decomposition methods are commonly employed to reformulate a PDE boundary-value problems onto a set of subdomains to make the solution more efficient on parallel machines as briefly outlined in section 4.2. Domain Decomposition methods are generally applicable to any discretization method for PDEs such as finite differences, finite volumes, spectral elements, etc., and have found widespread use in computational science and engineering[86].

Methods for the parallel solution of initial value PDE and ODE have only in recent years started to gain traction due to the seemingly ever increasing number of cores in new generations of supercomputers. The topic is however not at all new. As noted in a recent paper by Martin Gander titled “50 years of time parallel time integration”[124], the first paper to consider a pure time decomposition for the parallel solution of an ODEs was published as early as 1964 by Jürgen Nievergelt[232]. Nievergelt suggested that computers might be getting closer to the maximum limit at which they could operate, and wrote in the paper abstract on parallel machines that

> “To take full advantage for real-time computations of highly parallel computers as can be expected to be available in the near future, much of numerical analysis will have to be recast in a more "parallel" form. By this is meant that serial algorithms ought to be replaced by algorithms which consist of several subtasks which can be computed without knowledge of the results of the other subtasks. As an example, a method is proposed for "parallelizing" the numerical integration of an ordinary differential equation, which process, by all standard methods, is entirely serial.”

Unbeknown to the author at the time, each new generation of microchips would continue to demonstrate improved capacity for sequential computing for another 40 years.

The remainder of this chapter is dedicated to introducing Parallel-in-Time integration techniques, this with particular emphasis on Parareal. In section 5.1, a brief motivation and overview of methods in general is given, and in section 5.2 the Parareal algorithm is introduced. The chapter concludes with a brief summary of novel contributions in Part II of the thesis.

5.1 Parallel-in-Time Integration for PDEs

Solving time dependent PDEs is often done in a methods-of-line approach where the spatial components are discretized in some appropriate manner and a numerical integration technique applied to advance in time. The approach extends to distributed memory machines typically by applying some form of domain decomposition in space, letting independent nodes communicate boundary information of their local subdomains. The limitation to the approach lies in the strong scaling limit, i.e., increasing the number of nodes for a fixed problem size to achieve a reduction in time to compute.

One might naively expect to “run out of parallelism” - i.e. as the combined number of cores become sufficiently high, there are simply not enough degree’s of freedom for all cores to work all the time. However, solving a problem with millions, even upwards of a billion, degree’s of freedom in space may today be done on a potent workstation.

Conversely, even the largest available clusters have no more than a few million cores, everything included. This scaling limit is therefore somewhat theoretical, and not yet of much relevance for practitioners. So why does obtainable speed-up saturate in the strong-scaling limit? Consider a three dimensional domain in space divided into a number of quadratic sub-domains with n elements spanning each dimension. The compute work in each sub-domain is proportional to n^3. The boundary information that needs to be exchanged with neighboring sub-domains is proportional to n^2. As $n \to 1$, compute nodes will increasingly be spending time communicating boundary information rather than computing.

This particular limit is very much of practical concern. Moving a double between two individual compute nodes in a cluster is many orders of magnitude more expensive than a compute operation in terms of both wall-time and energy consumption. On large machines comprising thousands of nodes, this is a substantial bottleneck preventing applications of scaling efficiently, and therefore new algorithmic developments are required. A potential new path to obtain scaling beyond what is possible using conventional methods, is to introduce parallelism in the time integration procedure. Once a system of partial differential equation have been reduced to a large system of ordinary differential equations to be integrated over time, the problem is usually viewed as a sequential process. Many methods for parallel-in-time time integration have, however, been developed over the years. The earliest method to achive this trough a decomposition of the time domain was due to Nievergalt[232], which served as a precursor for several shooting type methods of which Parareal has emerged as the most popular approach. Parareal allows for potentially very large scale parallelism, the algorithm is non-invasive, and it achieves super linear convergence on certain problems. A main limitation of the approach is the difficulty of achieving high parallel efficiency and issues of stability on convection dominated problems as will be discussed in section5.2.

In recent years space-time multi-grid methods have been gaining traction as another potentially highly scalable approach. The first full space-time multigrid method was introduced in [302], and several others have since been proposed[211, 107, 127]. The methods may in some sense be seen as a generalization of methods such as Parareal, as they are designed for use with multiple grids and various ways of cycling between grids. In [128] it was shown how Parareal, under certain circumstances, may be seen as a two-level multigrid method. The disadvantage of space-time multigrid methods is that they compute on the entire space-time domain simultaneously, the added dimension means a substantial increase in memory consumption. In addition, numerical experiments appear to indicate that these methods experience convergence and stability issues when applied to convection dominated problems as well.

The methods mentioned thus far are all iterative. They exchange an increase in computational work, compared to the sequential algorithm, for the availability of parallelism in the otherwise sequential solution procedure. Direct solvers for parallel-in-time integration exists as well. Small scale parallelism suitable for node-level multi-core architectures can be achieved in predictor-corrector methods where the steps may be computed in parallel as demonstread in [213], or by deriving Runge Kutta schemes with independent stages as in[105]. RIDC (Revisionist Integral Deferred Correction) is a more recent development[71], here Integral Deferred Correction methods are modified in such a way that pipelining becomes possible, again leading to the availability of small-scale parallelism.

The remainder of this chapter will focus on the Parareal method and serve as an introduction to the content presented in chapters 6 to 8 and 10. For a comprehensive overview of past and present research directions on Parallel-in-Time integration techniques, we refer to[124].

5.2 The Parareal Method

The Parareal method has received substantial attention over the past decade. The method, first proposed in [192], borrows ideas from spatial domain decomposition to construct an iterative approach to solve the temporal problem in a parallel global-in-time approach. To present the method, consider a problem on the form

$$\begin{cases} \frac{d\mathbf{u}}{dt} + \mathcal{A}\left(t, \mathbf{u}\right) = 0 \\ \mathbf{u}\left(T_0\right) = \mathbf{u}_0 \quad t \in \left[T_0, T\right] \end{cases} \tag{5.1}$$

where $\mathcal{A} : \mathbb{R} \times V \to V'$ is a general operator depending on $\mathbf{u} : \Omega \times \mathbb{R}^+ \to V$ with V being a Hilbert space and V' its dual. Now, assume there exists a unique solution $\mathbf{u}\left(t\right)$ to (5.1) and decompose the time domain into n_t individual time slices

$$T_0 < T_1 < \cdots < T_{n_t - 1} < T_{n_t} = T. \tag{5.2}$$

Let $T_n = n\Delta T$. We now define an accurate solution operator $\mathcal{F}_{\Delta T}^{t}$ that, for any $t > T_0$, operates on a solution state $\mathbf{u}\left(t\right)$ and advance it Δ in the sense

$$\mathcal{F}_{\Delta T}^{T_n}\mathbf{u}\left(T_n\right) = \mathbf{U}_{T_n + \Delta T} \approx \mathbf{u}\left(T_n + \Delta T\right) \tag{5.3}$$

To solve (5.1) on $[T_0, T_0 + n\Delta T]$, define the operator $M_{\mathcal{F}}$

$$M_{\mathcal{F}} = \begin{bmatrix} 1 & & & \\ -\mathcal{F}_{\Delta T}^{T_0} & \ddots & & \\ & \ddots & \ddots & \\ & & -\mathcal{F}_{\Delta T}^{T_{n_t - 1}} & 1 \end{bmatrix} \tag{5.4}$$

with $\bar{\mathbf{U}} = [\mathbf{U}_0, \dots, \mathbf{U}_{n_t}]$ and $\bar{\mathbf{U}}_0 = [\mathbf{u}_0, 0, \dots, 0]$. The sequential solution procedure is equivalent to

$$M_{\mathcal{F}}\bar{\mathbf{U}} = \bar{\mathbf{U}}_0 \tag{5.5}$$

by forward substitution for $\bar{\mathbf{U}}$ to recover $\mathbf{U}_0 \cdots \mathbf{U}_{n_t}$ as approximations to $\mathbf{u}(T_0) \cdots \mathbf{u}(T_{n_t})$. If we instead seek to solve the system using a point-iterative approach, i.e., we seek the solution $\bar{\mathbf{U}}^{k+1} = \bar{\mathbf{U}}^k + \left(\bar{\mathbf{U}}_0 - M_{\mathcal{F}}\bar{\mathbf{U}}^k\right)$, we observe that at the beginning of each iteration, $\bar{\mathbf{U}}^k$ is known. In each iteration we may thus compute $\mathcal{F}^{T_1}_{\Delta T} \cdots \mathcal{F}^{T_{n_t}}_{\Delta T}$ on all intervals in parallel.

An important thing to note here is that the computational complexity of every iteration is strictly larger than that of the sequential solution procedure. Hence a reduced time to solution is possible only if the number of iterations k_{conv}, needed for convergence is much smaller than the number of time sub-domains n_t. To achieve this, a preconditioner is needed. Assuming the existence of some $M_{\mathcal{G}} \approx M_{\mathcal{F}}$, where $M_{\mathcal{G}}$ is computationally cheap, we can solve a preconditioned system on the form

$$(M_{\mathcal{G}})^{-1} M_{\mathcal{F}}\bar{\mathbf{U}} = (M_{\mathcal{G}})^{-1}\bar{\mathbf{U}}_0. \tag{5.6}$$

A natural approach to construct the above preconditioner $M_{\mathcal{G}}$ is to define a new operator $\mathcal{G}_{\Delta T}$ as with $\mathcal{F}_{\Delta T}$

$$\mathcal{G}_{\Delta T}\left(T_n, \mathbf{u}\left(T_n\right)\right) = \mathbf{U}_{T_n+\Delta T} \approx \mathbf{u}\left(T_n + \Delta T\right) \tag{5.7}$$

and relax the requirements on the accuracy of $\mathcal{G}_{\Delta T}$, by using a coarser grid or a different numerical model. Solving the system (5.6) iteratively using the standard preconditioned Richardson iterations we recover

$$\bar{\mathbf{U}}^{k+1} = \bar{\mathbf{U}}^k + (M_{\mathcal{G}})^{-1}\left(\bar{\mathbf{U}}_0 - M_{\mathcal{F}}\bar{\mathbf{U}}^k\right) \tag{5.8}$$

equivalent to

$$\begin{bmatrix} 1 & & & \\ -\mathcal{G}^{T_0}_{\Delta T} & \ddots & & \\ & \ddots & \ddots & \\ & & -\mathcal{G}^{T_{n_t-1}}_{\Delta T} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U}^{k+1}_0 \\ \mathbf{U}^{k+1}_1 \\ \vdots \\ \mathbf{U}^{k+1}_{n_t} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ \mathcal{F}^{T_0}_{\Delta T} - \mathcal{G}^{T_0}_{\Delta T} & \ddots & & \\ & \ddots & \ddots & \\ & & \mathcal{F}^{T_{n_t-1}}_{\Delta T} - \mathcal{G}^{T_{n_t-1}}_{\Delta T} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U}^{k}_0 \\ \mathbf{U}^{k}_1 \\ \vdots \\ \mathbf{U}^{k}_{n_t} \end{bmatrix} + \begin{bmatrix} \mathbf{u}_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{5.9}$$

from this we recover the Parareal algorithm in the form in which it is typically presented

$$\mathbf{U}^{k+1}_{n+1} = \mathcal{G}^{T_n}_{\Delta T}\mathbf{U}^{k+1}_n + \mathcal{F}^{T_n}_{\Delta T}\mathbf{U}^k_n - \mathcal{G}^{T_n}_{\Delta T}\mathbf{U}^k_n, \quad \mathbf{U}^0_{n+1} = \mathcal{G}^{T_n}_{\Delta T}\mathbf{U}^0_n, \quad \mathbf{U}^k_0 = \mathbf{u}(T_0). \tag{5.10}$$

Once the difference between two consecutive iterations is smaller than some tolerance, the algorithm is said to have converged. One must choose the tolerance small enough so that the error of the Parareal solution becomes as small as the error on a sequentially computed solution. A key point to note is how the number of iterations needed for convergence effect the parallel efficiency of the algorithm. Since in each iteration, computational work equivalent to the full sequential application of $\mathcal{F}_{\Delta T}$ n_t times, the upper bound on parallel efficiency scales as $1/k$ where k is the number of times the fine operators are computed in parallel accross n_t time-subdomains. In addition, fast convergence is not sufficient for efficient usage, it is essential to also consider the efficient distribution of parallel work. This has been the topic of several studies[101, 38, 9], and will be discussed in detail as part of the material presented in chapters 7 and 10.

A comprehensive introduction to Parareal may be found in [228], whilst important early contributions on the analysis of the method can be found in [288, 14, 128]. The algorithm has been applied to a wide range of applications such as simulation of dynamic physical models in automotive industry and powersystems [196, 147], optimal control[200, 207], time-fractional equations [313] and pricing of options [236].

The Parareal method has also been applied to plasma simulation with some success as presented in [262, 250]. Using various coarse operators, the authors achieve parallel-in-time speed-up, although with a low parallel efficiency of single digit percent whilst using up to 400 processors in time. In [248], convergence of the Parareal algorithm on lattice Boltzmann method applied to a laminar flow problem is presented. It is demonstrated that parallel speed-up is possible, albeit again with a low parallel efficiency, and no comparison with conventional domain decomposition in space is supplied. Numerical experiments of parallel-in-time integration using Parareal on the three-dimensional incompressible Navier-Stokes equations on a cavity problem is presented in [76]. The authors report that the space-time-parallel method can provide speedup beyond the saturation of a purely space-parallel approach. In the cavity test case, the performance saturates at a speedup of 18 with 32 cores in space. Using another 16 time-subdomains they report a combined space-time parallel speed-up of 27 using a total of 512 cores. Their results are in line with previous results, reporting low parallel efficiency on a limited time-domain, but nevertheless reach higher speedup than what is possible with the purely space-parallel approach. Similar results are reported in [211, 283], on a space-time parallel version of the Barnes-Hut tree code. The authors use PFASST for parallel-in-time integration and report scaling up to 262,144 cores on the IBM Blue Gene/P installation JUGENE, demonstrating that the space-time parallel code provides speedup beyond the saturation of the purely space-parallel approach.

Low parallel efficiency due to slow convergence of the algorithm is a general trend in past results. In [289] the authors present numerical experiments measuring the convergence of the Parareal algorithm when applied to the two dimensional Navier-Stokes equations on a driven cavity benchmark for different Reynolds numbers. They report that problems of instability and slow convergence increase with decreasing viscosity, i.e., when the flow becomes increasingly convection dominated. The effect is found to strongly depend on the spatial resolution of the problem. In [77] the authors present an analysis of the stability of Parareal applied to hyperbolic systems and convection dominated problems and it is shown that the instabilities are related to the regularity of the solution over time. They propose a stabilization scheme that modifies the iterative algorithm in such a way to avoid a transient phase of divergence before convergence. However convergence is observed to still be slow for long time-subdomains and the generalization of this approach is unclear. Other stabilizing schemes have been proposed [62, 61], but they suffer from similar limitations.

An important contribution to the understanding of how convergence is affected by the length of the time-interval to be integrated in parallel was presented in [126]. The authors show that for Hamiltonian systems, and a given problem with some coarse integrator, there exists a "window" in which time parallel integration is possible, and outside of which the method does not convergence. The author also demonstrate that convergence speed increase with smaller time-subdomains. In a recent paper [97], it is conjectured that there exists an optimal time-subdomain length at which convergence is rapid, yet the time-subdomain is still long enough that the communication of time-subdomain interfaces does not become a limiting factor to parallel speedup.

Recently there's been a renewed interest the application of Parareal to hyperbolic and convection dominated PDE problems. In early studies it was observed that the convergence rate of Parareal would deteriorate on certain problems as they become increasingly convection dominated, and several theoretical results were presented in support of these observations[288, 123]. A number of recent studies have however indicated that a PDE problem being convection dominated, or even hyperbolic, may not necessarily present an unsurmountable barrier for Parallel-in-time integration using Parareal[259, 41, 169].

5.3 Contributions in Part II

The remainder of Part II of the thesis consists of three chapters. In Chapter 6, we demonstrate a number of numerical experiments that serves to highlight findings in resent papers on the implication of phase errors, between the coarse and the fine operator, on the convergence rate of Parareal[169, 259]. The chapter acts, in part, as an introduction to Chapter 7, where a newly developed scheduler denoted CAAP is presented and used along with a phase-error free coarse operator to demonstrate space-time parallel speedup in excess of what is possible using space-parallel methods alone. Part II concludes with Chapter 8 in which a new method for creating Parallel-in-Time integration schemes is proposed. The method maintains the approach of using a coarse operator that is allowed to execute sequentially. The idea is to make an assumption on the difference between $\mathcal{F}$ and $\mathcal{G}$ when applied to a state $\mathbf{U}$, and based on the assumption derive an update equation to be used for extrapolation at time-subdomain interfaces. We demonstrate how that with this approach it is possible to derive methods that work for simple problems where the original Parareal algorithm fails spectacularly.

6 Numerical Experiments: Parareal Applied to Hyperbolic Problems

Several studies have demonstrated excellent convergence results for Parareal applied to diffusion dominated problems, both linear an non-linear of significant complexity[115, 266, 130, 70, 187]. The application of Parareal to hyperbolic and advection dominated problems has however remained a challenge since the inception of the algorithm in 2001[192, 108]. This is problematic as it prevents the usage of the algorithm for a large class of problems of practical interest such as fluid dynamics where issues of slow convergence or instabilities have been observed to arise with increasing Reynolds number[289]. The issue observed is that the algorithm either converges slowly, or experiences a transient phase of diverging before converging. Since the parallel efficiency is bounded by $1/k$, k being number of iterations needed for convergence, this is unacceptable.

Several analytical studies on the application of Parareal to hyperbolic problems have been published over the years. In [288], the behavior of Parareal as applied to a first order autonomous system of ODEs was considered. The authors were able to demonstrate that for systems with real eigenvalues, stability can be guaranteed, whereas if the eigenvalues of the system are purely imaginary, this is not the case. Parareal as applied to the advection equation has likewise been studied, and shown to be either unstable or inefficient for several different numerical discretizations[129, 123].

Many methods have been suggested to modify the algorithm in such a way to gaurentee stability. Unfortunately, all methods proposed thus far appear inadequate as they all suffer from substantial overhead, slow convergence and/or limited applicability[109, 77, 260, 62, 61]. Despite the many studies published on the topic, the question as to exactly how the instabilities arise has received little attention. In this chapter we present a number of numerical experiments to elucidate on this.

The chapter serves, in part, as a precursor to the material presented in Chapter 7. Chapter 7 is in essense an extended version of an already published paper[229]. In the paper, an asynchronous adaptive scheduler was presented, denoted CAAP, that balances various aspects of the execution process so to achieve higher parallel efficiency than other schedulers. The method was tested on the 2D nonlinear shallow water wave equation with a right hand side forcing term and moving boundaries for tsunami modeling, solved using a 3rd order space-time accurate WENO SSP-RK scheme. Trough large-scale experiments it was demonstrated not only that time-parallel speedup was possible, but that higher speedup was possible than what could be achieved using spacial domain-decomposition alone. The result is somewhat surprising as the general conviction in the community is that Parareal does not work for hyperbolic problems. The PDE being solved is purely hyperbolic, and the solution contains shock formation and waves interacting in a non-linear manner. In addition, due to the use of a high order WENO scheme, not much artificial numerical dissipation is introduced.

The results of the paper was presented at a talk at the 6th Conference on Parallel-in-Time Integration (PINT6) held in Monte Veritá Switzerland October 2017. At the Q/A following the presentation, the audience were primarily interested in figuring out why the algorithm converged at all, and only to a lesser extent interested in the asynchronous adaptive scheduler. It was suggested by a member of the audiance that a possible reason for why convergence was observed could be due to the specific choice of coarse operator. For the numerical experiments presented in the paper, a lower order Approximate Riemann solver was used as the coarse solver, applied to the same grid as the fine WENO SSP-RK scheme. The approximate Riemann solver captures wave propagation very well, so the difference with respect to the WENO SSP-RK is almost exclusively that it is more dissipative. In a recent study, it was shown trough a discrete dispersion analysis of the Parareal integration method as applied to a linear system of ODEs, that the source of instability is different discrete phase speeds on the coarse and fine level, and that the instability is particularly pronounced for higher wave modes[259]. This observation could potentially explain the convergence results presented in section 7.2.2 of Chapter 7.

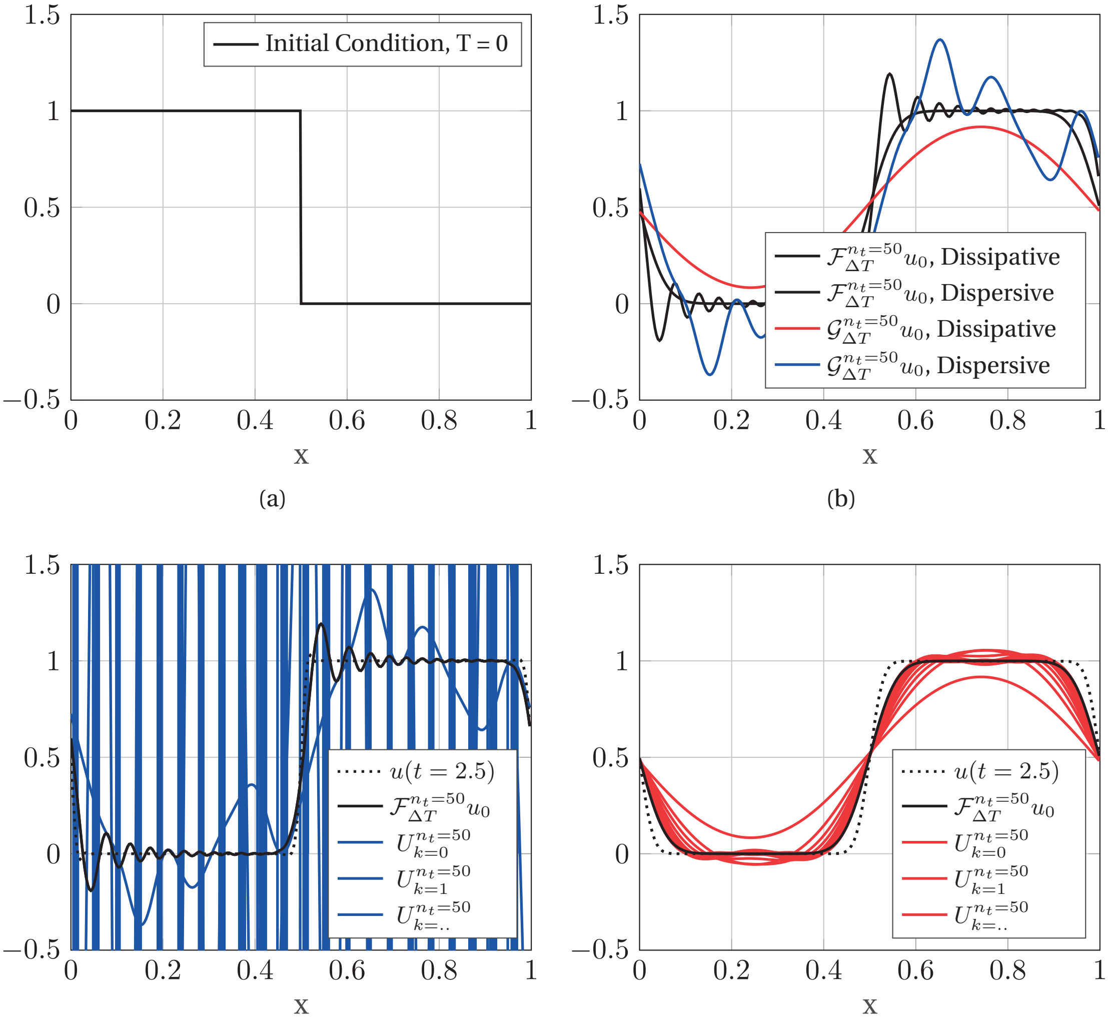
The remainder of this chapter is dedicated to verifying that this is indeed the case, and to demonstrate how instabilities develop due to phase speed difference between the coarse and fine operator.

6.1 The impact of Phase and Amplitude

In this section we present a few numerical experiments of Parareal applied to an advection dominated 1D advection-diffusion equation. Two experiments are made, one using a coarse operator that introduces only dispersive errors, and another using a coarse operator introducing only dissipative errors. We compute an approximation to the solution of the 1D advection-diffusion equation

$$\frac{\partial}{\partial t}u\left(x,t\right)+a\frac{\partial}{\partial x}u\left(x,t\right)=\kappa^{2}\frac{\partial^{2}}{\partial x^{2}}u\left(x,t\right) \tag{6.1}$$

on the domain $x \in [0,1]$ with periodic boundary conditions and $t \in [0,T]$. To act as a dispersive operator, a space-time 4th order accurate compact finite difference as proposed in [216] for the advection-diffusion equation is used. The dissipative operator is constructed by applying upwind finite difference to descritize the advection term, along with a centered finite difference stencil to descritize the diffusion term. Both terms treated are explicitly in time with a forward euler.



(a) (b)

(c) (d)

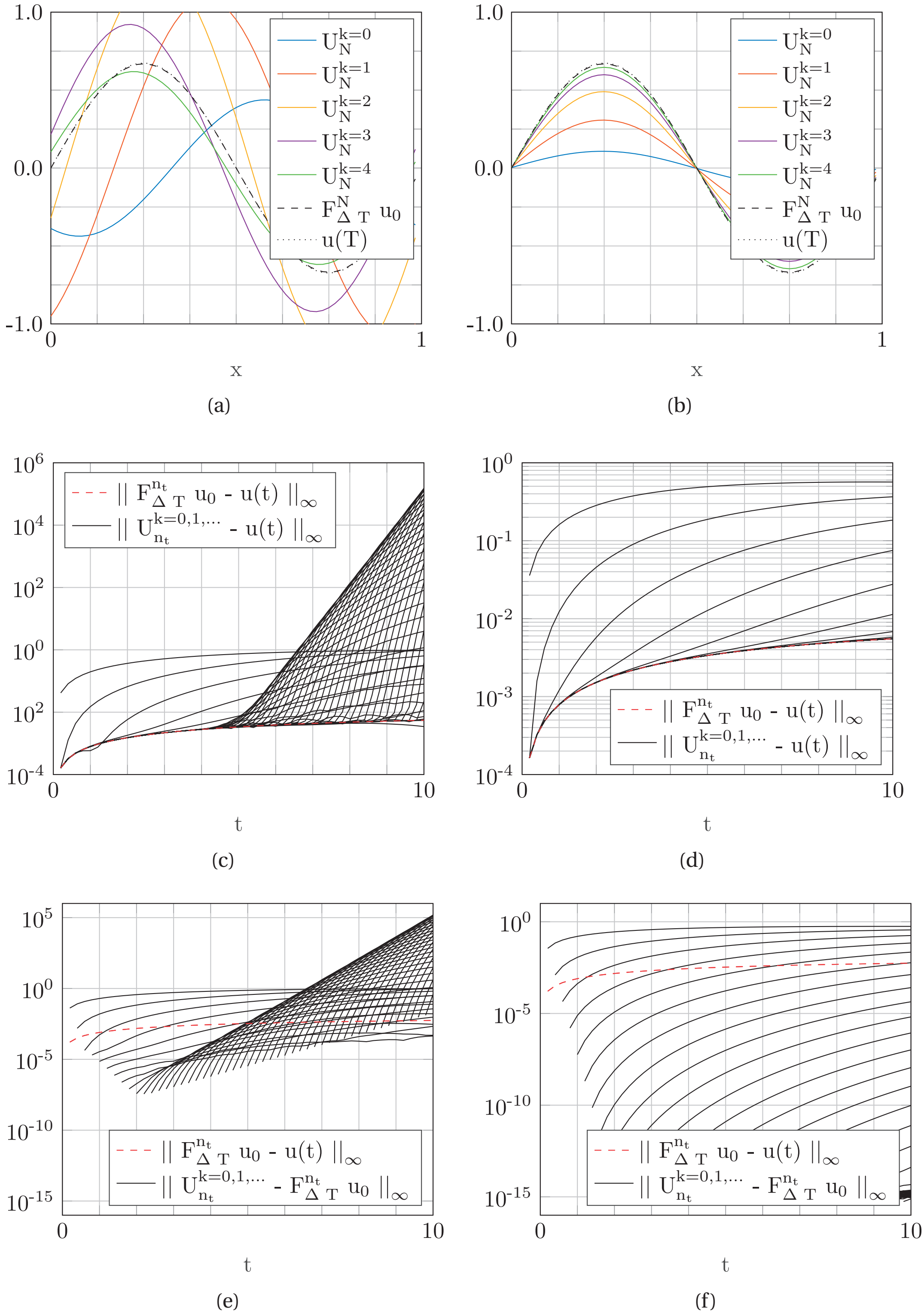
Figure 6.1 – The application of Parareal for parallel-in-time integration of the advection-diffusion equation with $\kappa = 10^{-5}$ and $a = 1$ to $T = 2.5$ using the initial condition in figure (a). The dispersive operators $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ are created using 301 and 31 points in space respectively, with a 4th order compact finite difference scheme. The dissipative operators $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ were created using a 1st order upwind scheme for the advection term and centered finite difference for the diffusion term, applied to a mesh with 601 and 61 points in space respectively.

Figure 6.1 depicts the result of an experiment using Parareal with $n_t = 50$ time-subdomains for the parallel-in-time integration of (6.1) with a discontinuous initial condition, as shown in figure 6.1a, and $\kappa = 10^{-5}$, $a = 1$, solved to $T = 2.5$, i.e. 2.5 wave periods. For $\mathcal{F}_{\Delta T}$, the dispersive operator is given 301 points in space, and the dissipative 601 points. In both cases $\mathcal{G}_{\Delta T}$ is constructed by factor 10 space-time coarsening and spline interpolation to move between grids. The difference between the two types of operators is demonstrated in figure 6.1b, where the result of the sequential application of each is plotted. When $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ are dispersive, the algorithm clearly becomes unstable as seen from figure 6.1c, but interestingly, when both operators are dissipative, figure 6.1d, the algorithm appears instead to converge monotonously.

Next we test the convergence behavior when combining the dispersive and dissipative operators. In figures 6.2 and 6.3, another set of numerical experiments are presented, this time including convergence plots. The error of the Parareal solution is measured with respect to both the sequential solution, and the exact solution, as a function of time for consecutive iterations. For the initial condition $u(x, 0) = \sin(2\pi x)$ is used, and the system is integrated over 10 wave periods using $n_t = 50$ time-subdomains. $\mathcal{F}_{\Delta T}$ is created using the same 4th order accurate compact finite difference stencil used for the previous set of experiments, here with 61 points in space. The dissipative coarse operator $\mathcal{G}_{\Delta T}$ is applied to the same grid with 61 points in space. The dispersive $\mathcal{G}_{\Delta T}$ is made by coarsening the space-time grid by a factor of 6, i.e. 11 points in space. The factor was chosen so to make the computational cost of the two coarse operators comparable.

In figure 6.2 the experiments are made using $\kappa = 10^{-3}$, and in figure 6.3 $\kappa = 10^{-5}$. In figures (a)-(b), the Parareal solution is plotted for the first 5 iterations at $T = 10$. In figures (c)-(d) the error with respect to the exact solution of the equation is given for each consecutive Parareal iteration as a function of time. In figures (e)-(f) the error with respect to the numerical approximation of the fine solution is given for each consecutive iterations. The difference is rather striking, one notes that when using the coarse operator that introduce wave phase errors, at $T = 10$, the algorithm diverges wildly and converges only towards the end when $K \to N$. When using the lower order operator as $\mathcal{G}_{\Delta T}$ that only introduce dissipative errors, the algorithm converges smoothly towards the right solution.

The results presented in figures 6.2(c)(e) and 6.3(c)(e) indicate that Parareal actually convergences for the dispersive coarse operator, for the first few iterations, for $T < 4$, i.e. less than 20 time-subdomains, before eventually diverging. However, comparing the two figures, it is clear that decreasing κ makes the interval for which it converges smaller. Crucially, this behavior is not seen in figures 6.2(d)(f) and 6.3(d)(f), no divergent behavior is observed when using the dissipative coarse operator, even for $\kappa = 10^{-5}$.

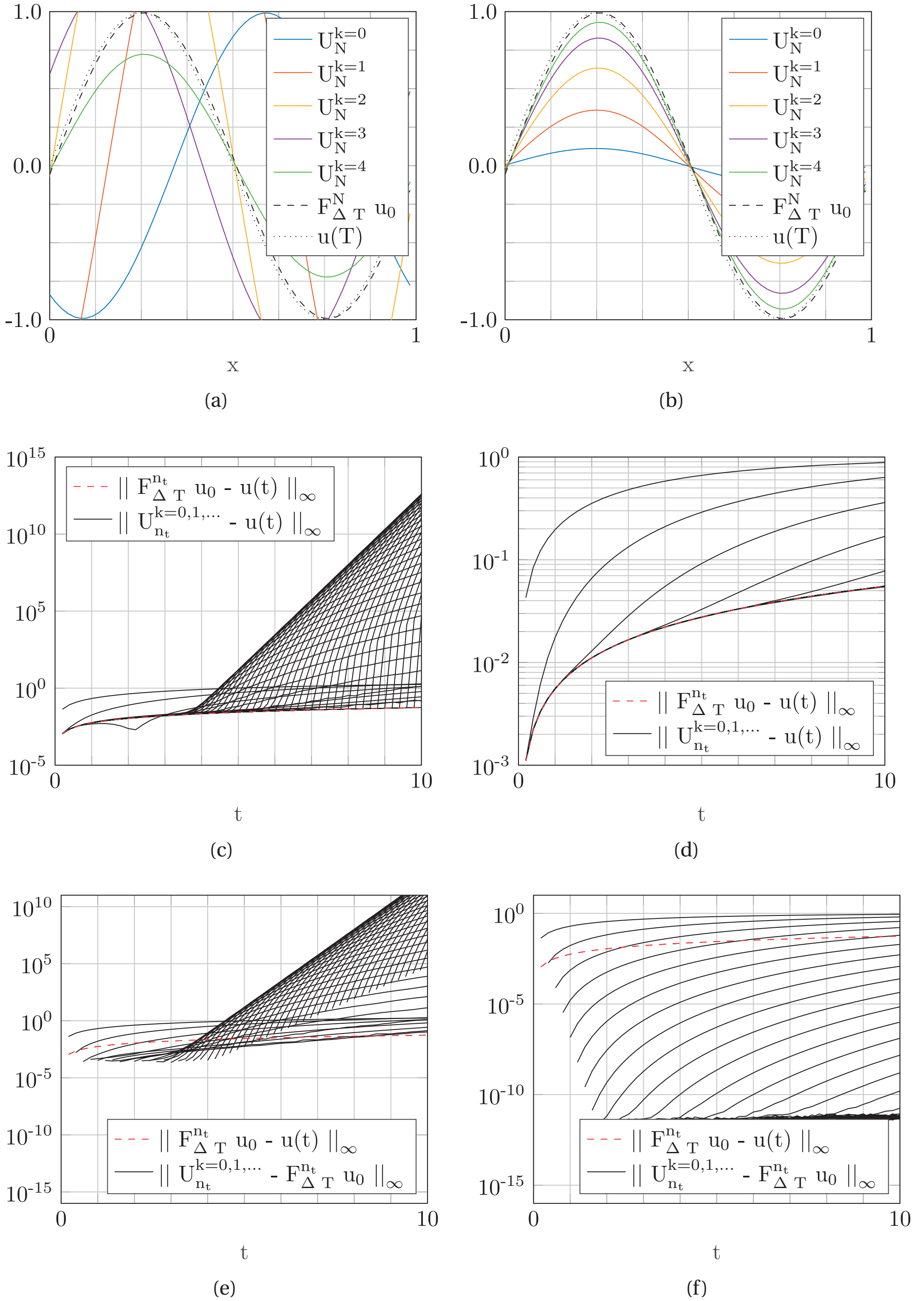


(a) (b)

(c) (d)

(e) (f)

Figure 6.2 – Convergence when computing an approximation to the solution of (6.1) with $\kappa = 10^{-3}$ and $a = 1$ to $T = 10$ on $n_t = N = 50$ time-subdomains. (a,c,e) using a factor 6 coarsened space-time grid as a dispersive $\mathcal{G}_{\Delta T}$. (b,d,f) using a lower order operator as a dissipative $\mathcal{G}_{\Delta T}$. In both cases $\mathcal{F}_{\Delta T}$ is the 4th order dispersive scheme.



(a) (b)

(c) (d)

(e) (f)

Figure 6.3 – Convergence when computing an approximation to the solution of (6.1) with $\kappa = 10^{-5}$ and $a = 1$ to $T = 10$ on $n_t = 50$ time-subdomains. (a,c,e) using a factor 6 coarsened space-time grid as a dispersive $\mathcal{G}_{\Delta T}$. (b,d,f) using a lower order operator as a dissipative $\mathcal{G}_{\Delta T}$. In both cases $\mathcal{F}_{\Delta T}$ is the 4th order dispersive scheme.

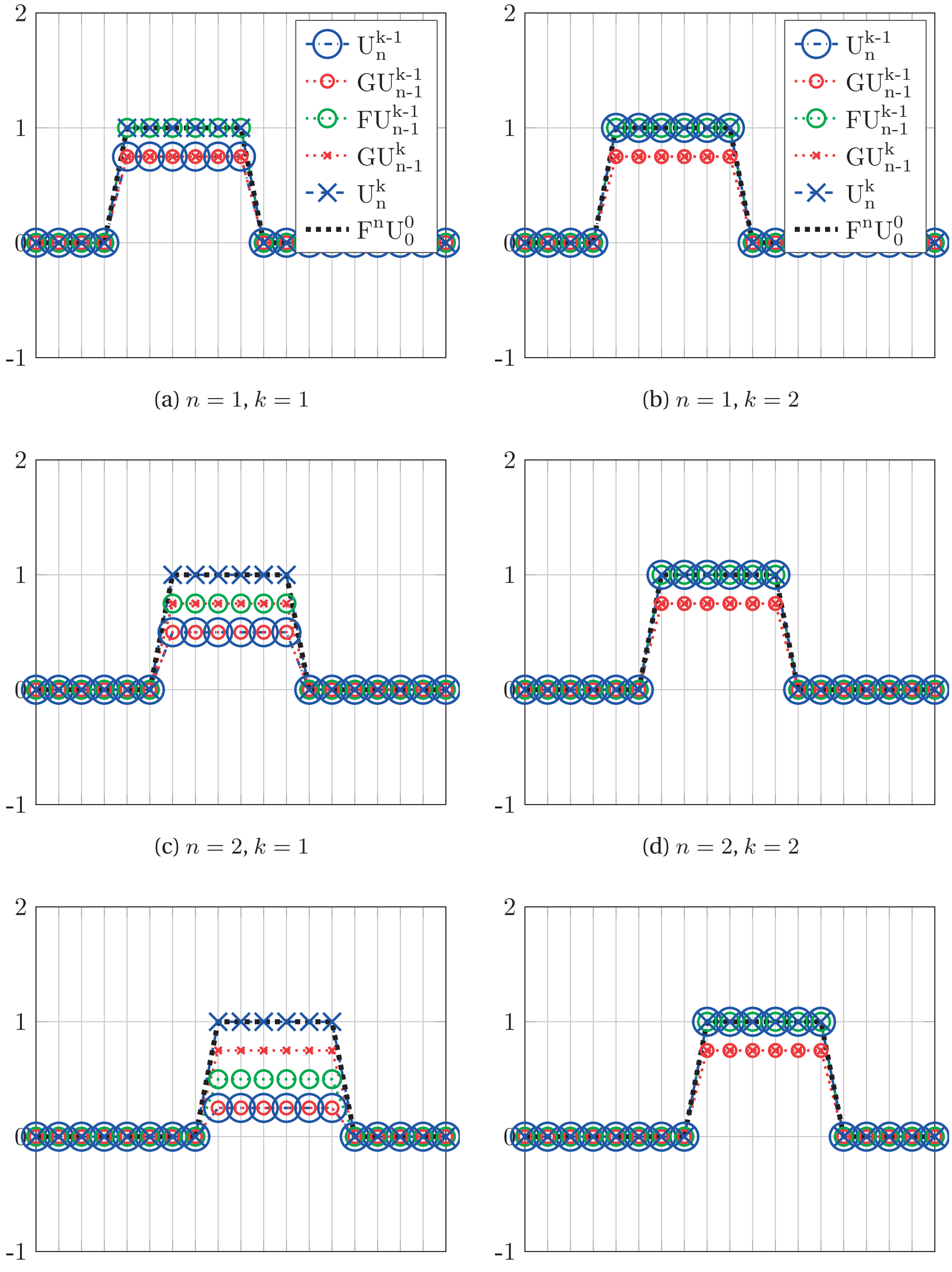
6.2 An Illustration of the Correction Procedure

The observations made in the numerical experiment presented in the previous section confirm the behavior suggested by the analysis presented in [259]; differences in wave phase speeds between the coarse and fine operator leads to instabilities whereas their absence leads to smooth convergence. The experiments and analysis referred to was made for a linear PDE and a linear system of ODEs respectively. The observations likewise appear to explain the, otherwise surprisingly, positive convergence results that will be presented in Chapter 7 for Parareal applied to a nonlinear hyperbolic system of PDEs in the form of the 2D shallow water wave equation with a right hand side forcing term.

The question still remains though; why does Parareal successfully correct dissipative errors on $\mathcal{G}_{\Delta T}$ with respect to $\mathcal{F}_{\Delta T}$, whereas dispersive errors tend to be amplified rather than corrected? To answer this question, and gain a deeper understanding of the correction procedure, we draw each step in the Parareal algorithm for different cases of coarse operator $\mathcal{G}_{\Delta T}$ behavior when applied to the scalar advection equation. The correction procedure is drawn for the first $n_t = 3$ intervals and $k = 2$ iterations, and 3 examples are given in figures 6.4, 6.5 and 6.6. In all figures, $\mathcal{F}_{\Delta T}$ is considered the true solution to the advection equation, and it advects a solution state two steps to the right. In figure 6.4, $\mathcal{G}_{\Delta T}$ is exactly $\mathcal{F}_{\Delta T}$, plus some constant amplitude error. In figures 6.5 and 6.6, $\mathcal{G}_{\Delta T}$ advects the solution a single step to the right instead of the two steps to the right of $\mathcal{F}_{\Delta T}$, i.e. a phase error.

The first thing to note from the experiment is how the addition of a constant phase error leads to the Parareal corrections becoming meaningless around the discontinuity as evident from following the correction procedure in figure 6.5. The horizontal movement of the wave is wrongly interpreted by Parareal as a need for vertical correction. Rather than correct errors, additional errors are introduced around the discontinuity. The same effect is seen in figure 6.6 on the pyramid shape, the softer angle change leads to a decrease in the magnitude in errors introduced, but does not eliminate them. Another important observation can be made from the correction procedure outlined in figure 6.4. When the error on $\mathcal{G}_{\Delta T}$ with respect to $\mathcal{F}_{\Delta T}$ is in the form of the addition of a constant, i.e. amplitude error, the algorithm converges exactly at iteration $k = 1$ for all n. This despite it being an advection equation with a moving discontinuity.

The correction that the algorithm performs fundamentally assumes that $\mathcal{G}_{\Delta T}$ differs from $\mathcal{F}_{\Delta T}$ by some constant function $c_n(x)$. If this is satisfied, the algorithm is exact for all n at the first correction $k = 1$. Regardless of the underlying properties of $\mathcal{F}_{\Delta T}$. This assumption is fundamentally flawed when $\mathcal{G}_{\Delta T}$ has dispersive error components that differ from $\mathcal{F}_{\Delta T}$. The horizontal movement of waves is wrongly interpreted by the algorithm as a need for correction, which results in the injection of additional high frequency error components. The magnitude of the problematic error components in-



(a) $n = 1$, $k = 1$

(b) $n = 1$, $k = 2$

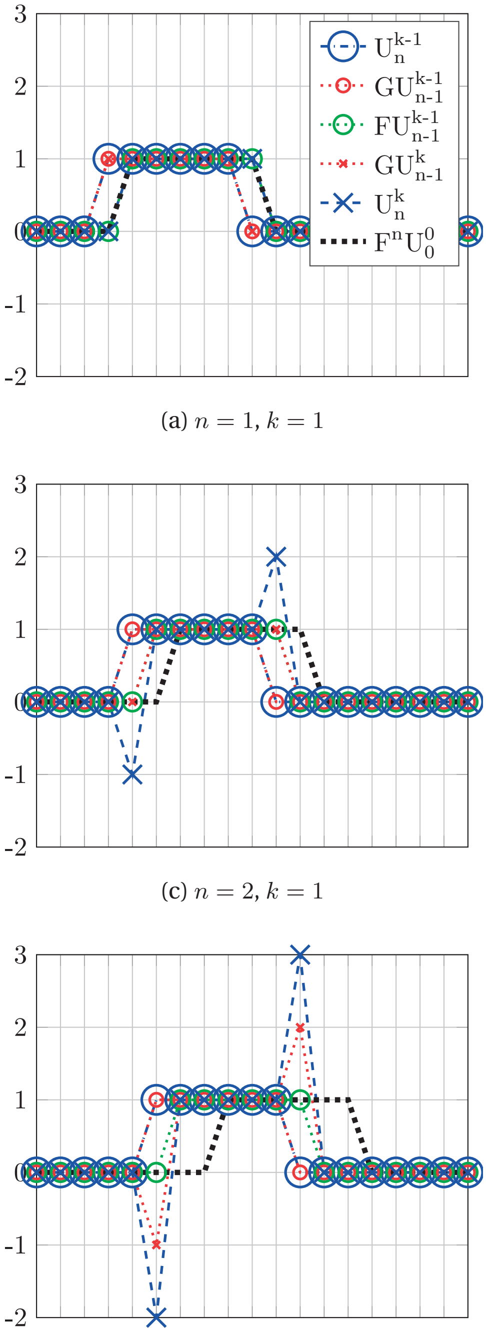
(c) $n = 2$, $k = 1$

(d) $n = 2$, $k = 2$

(e) $n = 3$, $k = 1$

(f) $n = 3$, $k = 2$

Figure 6.4 – Illustration of the Parareal correction procedure on a shock moving to the right with a constant velocity of two steps per ΔT. $\mathcal{F}_{\Delta T}$ is exact in that it takes any state U and moves it two steps to the right. The coarse operator $\mathcal{G}_{\Delta T}$ also moves the solution two steps to the right, but introduces an error on the amplitude such that $\mathcal{G}_{\Delta T}U_n^k = \mathcal{G}_{\Delta T}U_n^k - 0.25$ if $\mathcal{G}_{\Delta T}U_n^k \neq 0$ and $\mathcal{G}_{\Delta T}U_n^k = \mathcal{G}_{\Delta T}U_n^k$ otherwise. One notes that U_n^k for all n converge to the exact solution at $k = 1$.



(a) $n = 1, k = 1$

(b) $n = 1, k = 2$

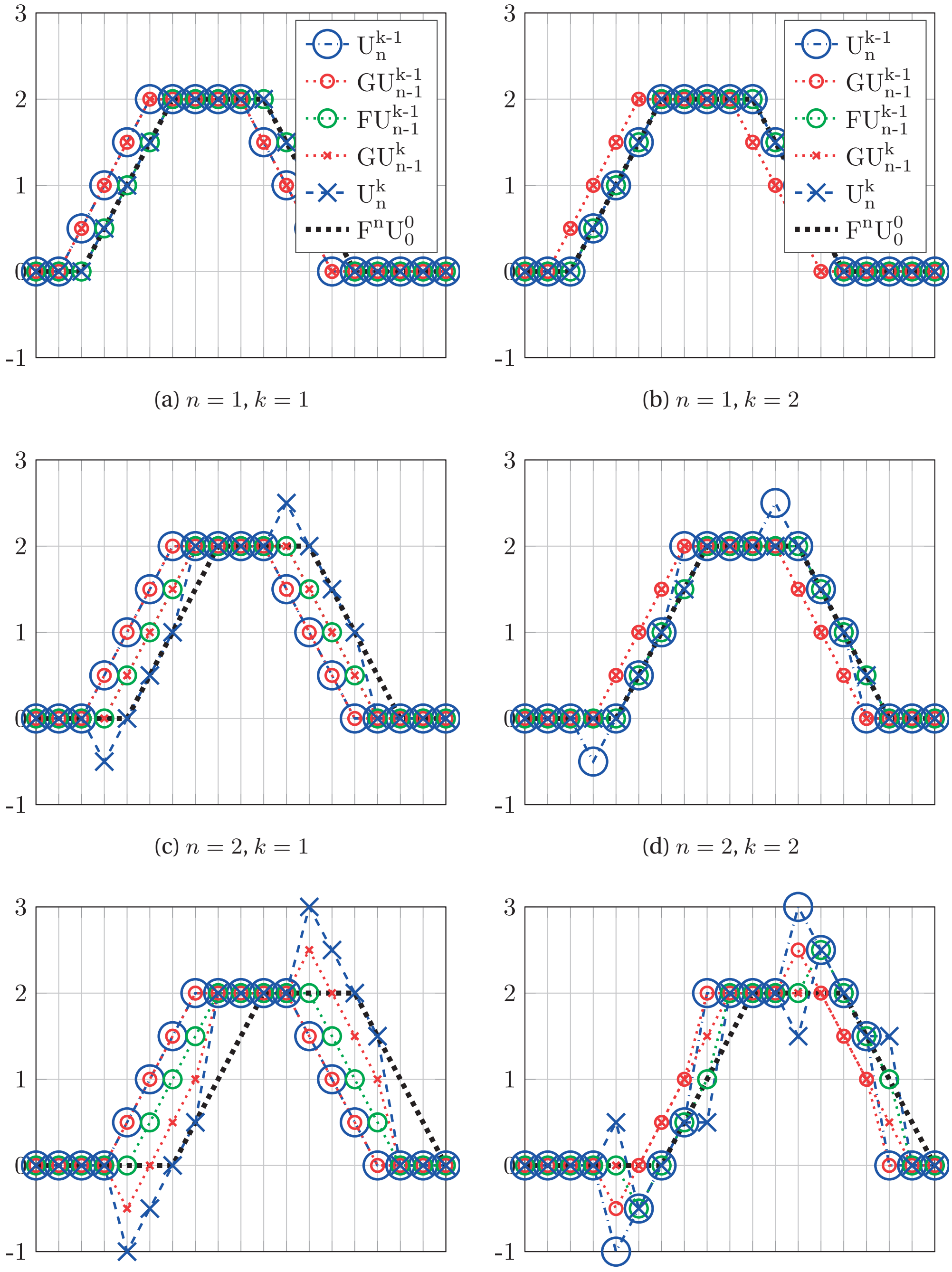
(c) $n = 2, k = 1$

(d) $n = 2, k = 2$

(e) $n = 3, k = 1$

(f) $n = 3, k = 2$

Figure 6.5 – Illustration of the Parareal correction procedure on a shock moving to the right with a constant velocity of two steps per ΔT. $\mathcal{F}_{\Delta T}$ is exact in that it takes any state U and moves it two steps to the right. The coarse operator $\mathcal{G}_{\Delta T}$ introduces an error as it only moves a state U one step to the right for each ΔT. One notes that large oscillations are introduced around the shock and that no meaningful correction is made on the state U_n^k for all n such that $n > k$.



(a) $n = 1, k = 1$

(b) $n = 1, k = 2$

(c) $n = 2, k = 1$

(d) $n = 2, k = 2$

(e) $n = 3, k = 1$

(f) $n = 3, k = 2$

Figure 6.6 – Illustration of the Parareal correction procedure on a wave with sharp edges moving to the right with a constant velocity of two steps per ΔT. $\mathcal{F}_{\Delta T}$ is exact in that it takes any state U and moves it two steps to the right. The coarse operator $\mathcal{G}_{\Delta T}$ introduces an error as it only moves a state U one step to the right for each ΔT. One notes that for all points x_i for which it is true that $\frac{\partial}{\partial x}\mathcal{F}_{\Delta T}U_n^k\left(x\right)\big|_{x_i} = \frac{\partial}{\partial x}\mathcal{G}_{\Delta T}U_n^k\left(x\right)\big|_{x_i} = \frac{\partial}{\partial x}\mathcal{G}_{\Delta T}U_n^{k+1}\left(x\right)\big|_{x_i}$, the correction is exact already by the first correction $k = 1$. For the points x_i at which the angles differ, large oscillations are introduced and no meaningful correction made.

troduced by the correction is proportional to the frequencies already present, therefore, for the algorithm to be stable, there must be sufficient artificial or natural diffusion in the problem to dampen these errors components, otherwise they will be magnified in subsequent iterations.

6.3 Summary

Observing the correction procedure as applied to a pure advection equation exposes the limitations of Parareal. In the limit that the solution contain high frequencies, whilst $\mathcal{G}_{\Delta T}$ differ from $\mathcal{F}_{\Delta T}$ only in terms of phase errors, the Parareal correction is useless. The error of $\mathcal{G}_{\Delta T}$ with respect to $\mathcal{F}_{\Delta T}$ will be magnified rather than corrected. In this case, the algorithm is exact only for time-subdomains $n <= k$, for all $n > k$, errors are introduced and amplified in U_n^k. Thus, for problems where $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$ have different dispersive error components, it appears that it is only possible to use Parareal in one of two ways:

1. Ensure somehow that there is sufficient dissipation in the problem to dampen out all amplification by Parareal on dispersive error components.
2. Design a coarse operator $\mathcal{G}_{\Delta T}$ that is exactly $\mathcal{F}_{\Delta T}$ + some dissipation.

Option 1 is problematic for several reasons. First, no general theory exists to a priori predict when/if there is enough dissipation to stabilize the algorithm. Second, if one was to add additional artificial dissipation to $\mathcal{G}_{\Delta T}$ to dampen the errors introduced, the convergence rate will decrease since the dissipation can not distinguish between which features are due to errors and which are not. Option 2 may be a viable approach in certain cases. For the numerical solution of nonlinear conservation laws, many widely used schemes, such as Godunov method and various Approximate Riemann solvers, capture the speed of waves exactly, but are dissipative. Applications relying on discretizations of this type for the numerical solution of conservation laws may be a possible use-case for Parareal, despite the underlying problem being hyperbolic in nature.

In Chapter 7 we demonstrate one such example. Here Parareal is applied for the parallel-in-time integration of a conservation law in the form of 2D shallow water wave equation solved by WENO SSP-RK[279, 315]. A Roe approximate Riemann solver is used to create $\mathcal{G}_{\Delta T}$. The first order scheme captures the position of waves and shocks very well, but is dissipative compared to the higher order WENO SSP-RK scheme.

At the time of implementing the latter, the observations presented in this Chapter were not known. The choice of coarse operator was however not simply a lucky guess that just happened to work well. Before choosing to use that particular coarse operator, we had made a number of experiments on the 1D equation with different coarse operator candidates to determine which operator would work better. Among the candidates were using a simple coarsened space-time mesh, as well as various lower order schemes applied to the same mesh in the form of Lax-Friedrichs method, Roe's method[190] and two relaxation schemes[177]. A range of operators was tested because coarse operator efficiency is essential for the method to work well as will be discussed further in Chapter 7, so one should choose a coarse operator which is as accurate as possible under whatever cost constraint given. I.e., to create a coarse operator that is a factor of 10 or 100 faster in terms of computational complexity, simply coarsened the grid might not be most effective accurate option.

In this chapter we demonstrated how differences in dispersive error components between $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ are the cause of the instabilities and slow convergence often reported for Parareal applied to hyperbolic and convection dominated problems. In Chapter 8, we use the insight gained here to propose a new way of creating parallel-in-time integration schemes, inspired by Parareal, that allows for phase errors to be corrected upon.

7 Communication Aware Adaptive Parareal

In the strong scaling limit, the performance of conventional spatial domain decomposition techniques for the parallel solution of PDEs tend to saturate as briefly outlined in Section 4.1. When spatial sub-domains become small, the communication of boundary information and other overhead tend to dominate to the extend that it becomes a bottleneck for parallel acceleration. A potential path beyond this scaling limit is to introduce domain-decomposition in time. One such popular approach is the Parareal algorithm which has received substantial attention due to its generality and potential scalability.

Low efficiency, particularly on convection dominated problems, has, however limited the adoption of the method. In this chapter we demonstrate that it is possible to obtain time-parallel speedup on the non-linear shallow water wave equation, and that we may obtain parallel acceleration beyond what is possible using conventional spatial domain-decomposition techniques alone. Two factors are essential in achieving this. First, for Parareal to converge for the hyperbolic problem we use a finite volume method with a Roe numerical flux as the preconditioner. This coarse operator introduces only dissipative errors with respect to the 3rd order accurate WENO-RK discretization used to solve the PDE system.

The preconditoner is relatively expensive and convergence is slow unless the time-subdomains are short. We therefore introduce a new scheduler that we denote Communication Aware Adaptive Parareal (CAAP). CAAP increases obtainable speed-up by minimizing the time-subdomain length without making communication of time-subdomains too costly whilst also adaptively overlapping consecutive cycles of the Parareal to mitigate the impact of a relatively expensive coarse operator. The content of this chapter has been published in a condensed form in [229].

7.1 Introduction

The rapid evolution of computers used to model physical phenomena in the computational sciences poses new challenges for algorithms. The growing number of cores, the increasingly convoluted cache hierarchies, and the use of accelerators all seek to boost the computational capacity of individual nodes. At the same time, the number of compute nodes in distributed memory machines has increased dramatically. This development towards increasing hardware parallelism exposes algorithmic shortcomings and requires a rethinking of the fundamental algorithms to maintain scalability and enable efficient use of the computing platform [92]. The focus of this chapter is on the Parareal method that has received substantial attention over the past decade. The Parareal method, first proposed in [192], borrows from ideas in (spatial) domain decomposition to construct an iterative approach to solve the temporal problem in a parallel-in-time approach. An introduction to the method was given in chapter 5, and in chapter 6 the application of the algorithm was investigated in the context of linear hyperbolic and convection dominated PDEs, which serves in part as an introduction to chapter 7. A comprehensive introduction to parallel-in-time integration schemes in general can be found in [124], and an introduction to Parareal specifically in [228].

The chapter begins with an introduction to the nonlinear shallow water wave equation and to the numerical scheme used to solve it, this followed by the introduction of the test case used. In section 7.4 we present theoretical considerations on how to a priori estimate the optimal time-subdomain length for the time-decomposition. To effectively decouple the time-subdomain length and the total time to be integrated with Parareal, we introduce an adaptive Parareal variant in section 7.3 based on the scheduler introduced in [9]. The approach allows consecutive Parareal cycles to overlap in time to balance processor utilization and convergence efficiently. Henceforth, we will denote the combination of the adaptive work scheduler and an informed choice on the time-subdomain length as CAAP, an abbreviation of Communication-aware Adaptive Parareal. The shallow water wave equation test case is introduced in Section 7.2 along with relevant notation, and the numerical experiments and scaling tests that demonstrate the performance of our approach are presented in section 7.5.

7.2 2D Shallow Water Equation with Explicit SSP-RK WENO

The shallow water wave equation is used to model a wide array of wave phenomena. Simulation of trans-ocean waves, flows in rivers and coastal areas, hydraulic engineering, and atmospheric modeling are among the many examples of application. The system of coupled partial differential equations that constitutes the shallow water wave equation is nonlinear and purely hyperbolic. The equation captures fundamental phenomena across different scales in space and time including shocks that may form during the solutions procedure even for perfectly smooth initial conditions. The equation is therefore a challenging case for parallel-in-time integration and, as such, an excellent platform for measuring to what extend time-parallel integration is possible for hyperbolic problems. The two dimensional version of the equation may be written as

$$\begin{cases} h_t + (hu)_x + (hv)_y = 0 \\ (hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x + (huv)_y = -ghz_x \\ (hv)_t + (huv)_x + \left(hv^2 + \frac{1}{2}gh^2\right)_y = -ghz_y \end{cases} \tag{7.1}$$

where $h := h\,(x, y, t)$ denotes the water height, $u := u\,(x, y, t)$ and $v := v\,(x, y, t)$ the velocity in the x and y direction, respectively. $z := z\,(x, y)$ denotes overland topography and underwater bathymetry whilst g denotes the gravitational constant. The equation is often presented in conservation form as

$$q_t + f\,(q)_x + g\,(q)_y = s\,(h, z) \tag{7.2}$$

where $q = (h, hu, hv)^T$, and the two flux functions $f\,(q)$, $g\,(q)$ are given by

$$f\,(q) = \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix},\ g\,(q) = \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix} \tag{7.3}$$

The source term $s\,(h, z)$ is needed for non-flat bathymetrys. For inundation modeling, the source term occationally also include Manning's law, an empirically derived friction term that is added to better capture the physics of land-overflow. In the code the be accelerated, a simple thin-layer mesh reduction technique is used for inundation modelling, but no friction terms are added as hydrological studies is not the primary concern of this work. We refer to the works of [201, 315] for an introduction to the shallow water wave equation and inundation modelling.

7.2.1 Introducing the Operators

Finite volume schemes are a popular approach for solving hyperbolic conservation laws as the underlying physics is represented in a natural way. Let $I_{i,j} = \left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right] \times \left[y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}\right]$ define a structured rectangular uniform mesh. In a finite-volume scheme, we seek the cell average

$$Q_{ij}\,(t) = \frac{1}{\Delta x \Delta y} \int_{I_{i,j}} q\,(x, y, t)\, dxdy \tag{7.4}$$

that approximates $q\,(x, y, t)$ at every cell $I_{i,j}$ for a given time-step t. The spatial derivatives must therefore be approximated using cell-averages. To do so, a Weighted Essentially Non-Oscillatory (WENO) scheme is used for the reconstruction along with a

Strong-Stability-Preserving type explicit Runge-Kutta scheme (SSP-ERK) to integrate the resulting system of coupled ODEs. This scheme is 3rd order accurate in both space and time. To use Parareal, a "coarse" operator $\mathcal{G}_{\delta T}$ is required that act as a preconditioner in the Parareal iteration. To construct such a preconditioner, we use a simple finite volume scheme with an approximate Riemann solver, first order in time and space method. This coarse operator is introduced in Section 7.2.1, followed by the introduction of the 3rd order WENO SSP-ERK scheme in Section 7.2.1. In both cases, the methods are presented in their complete form, but without derivation and analysis. We refer to [279] for an introduction to ENO and WENO based methods, and [315] for higher order WENO applied to the shallow water wave equation with wetting and drying of cells. For both the WENO SSP-ERK and the lower order method the integration procedure is adaptive, in each step the longest possible time-step that satisfy the CFL condition for all cells is computed and used. Figure 7.4 contains a visualization of a Tsunami wave hitting the coastline of the main islands of Hawaii as simulated with the finite-volume discretization of the inhomogeneous shallow water wave equation (7.1) solved using the WENO SSP-ERK scheme. Following the next two sections where the numerical scheme is briefly outlined, the test-case that will be used to evaluate our proposed method for parallel-in-time integration is presented.

Roe's Method

For the coarse operator $\mathcal{G}_{\Delta T}$ used to solve (7.1), we use a standard finite volume method with an approximate Riemann solver[253]. The method is explicit, first order in time and space. The complete scheme is outlined in (7.5)-(7.16). Here we omit any details on derivation and analysis and instead refer to [190] for a comprehensive introduction to finite volume methods. Henceforth we will refer to the scheme as "Roe's Method" following the convention used in [190]. To find Q_{ij}^{n+1} from Q_{ij}^{n+1}, one evaluates

$$Q_{ij}^{n+1} = Q_{ij}^{n} - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2},j}^{n} - F_{i-\frac{1}{2},j}^{n}\right) - \frac{\Delta t}{\Delta y}\left(G_{i,j+\frac{1}{2}}^{n} - G_{i,j-\frac{1}{2}}^{n}\right) + \frac{1}{\Delta t}s\left(Q_{i,j}^{n+1}\right) \tag{7.5}$$

where

$$F_{i-\frac{1}{2},j}^{n} = \frac{1}{2}\left(f\left(Q_{i-1,j}^{n}\right) - f\left(Q_{i,j}^{n}\right)\right) - \frac{1}{2}\left|\tilde{\mathbf{J}}_{i-\frac{1}{2},j}^{f}\right|\left(Q_{i,j}^{n} - Q_{i-1,j}^{n}\right) \tag{7.6}$$

$$G_{i,j-\frac{1}{2}}^{n} = \frac{1}{2}\left(g\left(Q_{i,j-1}^{n}\right) - g\left(Q_{i,j}^{n}\right)\right) - \frac{1}{2}\left|\tilde{\mathbf{J}}_{i,j-\frac{1}{2}}^{g}\right|\left(Q_{i,j}^{n} - Q_{i,j-1}^{n}\right) \tag{7.7}$$

Here s is the RHS function taking into account bathymetry. Due to the special structure of $s\left(Q\right)$, (7.5) may be evaluated explicitly, see (7.1). In the above, $\tilde{\mathbf{J}}_{i-\frac{1}{2},j}^{f}$ and $\tilde{\mathbf{J}}_{i,j-\frac{1}{2}}^{g}$ are derived from the Jacobian matrices of $f\left(q\right)$ and $g\left(q\right)$ respectively. From (7.3) we see that

$$\nabla_q f(q) = \begin{bmatrix} 0 & 1 & 0 \\ -u^2+gh & 2u & 0 \\ -uv & v & u \end{bmatrix}, \quad \nabla_q g(q) = \begin{bmatrix} 0 & 0 & 1 \\ -uv & v & u \\ -v^2+gh & 0 & 2v \end{bmatrix} \tag{7.8}$$

The matrices $\tilde{\mathbf{J}}^f_{i-\frac{1}{2},j}$ and $\tilde{\mathbf{J}}^g_{i,j-\frac{1}{2}}$ are then defined as the Jacobian matrices (7.8) evaluated at the Roe averages, defined by

$$\tilde{u}_{i-\frac{1}{2},j} = \frac{\sqrt{h_{i-1,j}}u_{i-1,j} + \sqrt{h_{i,j}}u_{i,j}}{\sqrt{h_{i-1,j}} + \sqrt{h_{i,j}}}, \quad \tilde{v}_{i-\frac{1}{2},j} = \frac{\sqrt{h_{i-1,j}}v_{i-1,j} + \sqrt{h_{i,j}}v_{i,j}}{\sqrt{h_{i-1,j}} + \sqrt{h_{i,j}}} \tag{7.9}$$

for $\tilde{u}_{i-\frac{1}{2},j}$, $\tilde{v}_{i-\frac{1}{2},j}$ and

$$\tilde{u}_{i,j-\frac{1}{2}} = \frac{\sqrt{h_{i,j-1}}u_{i,j-1} + \sqrt{h_{i,j}}u_{i,j}}{\sqrt{h_{i,j-1}} + \sqrt{h_{i,j}}}, \quad \tilde{v}_{i,j-\frac{1}{2}} = \frac{\sqrt{h_{i,j-1}}v_{i,j-1} + \sqrt{h_{i,j}}v_{i,j}}{\sqrt{h_{i,j-1}} + \sqrt{h_{i,j}}} \tag{7.10}$$

for $\tilde{u}_{i,j-\frac{1}{2}}$ and $\tilde{v}_{i,j-\frac{1}{2}}$. The Roe averages on the water height cell average $\tilde{h}_{i-\frac{1}{2},j}$ and $\tilde{h}_{i,j-\frac{1}{2}}$ are given by

$$\tilde{h}_{i-\frac{1}{2},j} = \frac{1}{2}\left(h_{i-1,j} + h_{i,j}\right), \quad \tilde{h}_{i,j-\frac{1}{2}} = \frac{1}{2}\left(h_{i,j-1} + h_{i,j}\right) \tag{7.11}$$

The Jacobian matrices (7.8) have the following two eigensystem decompositions

$$\Lambda^f = \begin{bmatrix} u & & \\ & u-\sqrt{gh} & \\ & & u+\sqrt{gh} \end{bmatrix}, \quad R^f = \begin{bmatrix} 0 & 1 & 1 \\ 0 & u-\sqrt{gh} & u+\sqrt{gh} \\ 1 & v & v \end{bmatrix} \tag{7.12}$$

$$\Lambda^g = \begin{bmatrix} v & & \\ & v-\sqrt{gh} & \\ & & v+\sqrt{gh} \end{bmatrix}, \quad R^g = \begin{bmatrix} 0 & 1 & 1 \\ 1 & u & u \\ 0 & v-\sqrt{gh} & v+\sqrt{gh} \end{bmatrix} \tag{7.13}$$

from which we define

$$\begin{aligned} \tilde{\Lambda}^f_{i-\frac{1}{2},j} &= \begin{bmatrix} \tilde{u}_{i-\frac{1}{2},j} & & \\ & \tilde{u}_{i-\frac{1}{2},j} - \tilde{c}_{i-\frac{1}{2},j} & \\ & & \tilde{u}_{i-\frac{1}{2},j} + \tilde{c}_{i-\frac{1}{2},j} \end{bmatrix}, \\ \tilde{R}^f_{i-\frac{1}{2},j} &= \begin{bmatrix} 0 & 1 & 1 \\ 0 & \tilde{u}_{i-\frac{1}{2},j} - \tilde{c}_{i-\frac{1}{2},j} & \tilde{u}_{i-\frac{1}{2},j} + \tilde{c}_{i-\frac{1}{2},j} \\ 1 & \tilde{v}_{i-\frac{1}{2},j} & \tilde{v}_{i-\frac{1}{2},j} \end{bmatrix} \end{aligned} \tag{7.14}$$

and

$$\begin{aligned}\tilde{\Lambda}^g_{i,j-\frac{1}{2}} &= \begin{bmatrix} v_{i,j-\frac{1}{2}} & & \\ & \tilde{v}_{i,j-\frac{1}{2}} - \tilde{c}_{i,j-\frac{1}{2}} & \\ & & \tilde{v}_{i,j-\frac{1}{2}} + \tilde{c}_{i,j-\frac{1}{2}} \end{bmatrix}, \\ \tilde{R}^g_{i,j-\frac{1}{2}} &= \begin{bmatrix} 0 & 1 & 1 \\ 1 & \tilde{u}_{i,j-\frac{1}{2}} & \tilde{u}_{i,j-\frac{1}{2}} \\ 0 & \tilde{v}_{i,j-\frac{1}{2}} - \tilde{c}_{i,j-\frac{1}{2}} & \tilde{v}_{i,j-\frac{1}{2}} + \tilde{c}_{i,j-\frac{1}{2}} \end{bmatrix}\end{aligned} \tag{7.15}$$

where $\tilde{c}_{i-\frac{1}{2},j} = \sqrt{g\tilde{h}_{i-\frac{1}{2},j}}$ and $\tilde{c}_{i,j-\frac{1}{2}} = \sqrt{g\tilde{h}_{i,j-\frac{1}{2}}}$, so that $\left|\tilde{\mathbf{J}}^f_{i-\frac{1}{2},j}\right|$ and $\left|\tilde{\mathbf{J}}^g_{i,j-\frac{1}{2}}\right|$ can be written as

$$\left|\tilde{\mathbf{J}}^f_{i-\frac{1}{2},j}\right| = R^f_{i-\frac{1}{2},j}\left|\Lambda^f_{i-\frac{1}{2},j}\right|\left(R^f_{i-\frac{1}{2},j}\right)^{-1}, \quad \left|\tilde{\mathbf{J}}^g_{i,j-\frac{1}{2}}\right| = R^g_{i,j-\frac{1}{2}}\left|\Lambda^g_{i,j-\frac{1}{2}}\right|\left(R^g_{i,j-\frac{1}{2}}\right)^{-1} \tag{7.16}$$

Together with equations (7.5), (7.6) and (7.7) this completes the description for Roe's method when applied to the 2D shallow water wave equation. Note that the evaluation of $s\left(Q^{n+1}_{i,j}\right)$ in (7.5) does not infer that Q^{n+1}_{ij} can not be computed explicitly for all i, j since $h^{n+1}_{i,j}$ can be evaluated explicitly using only Q^n_{ij} before computing $hu^{n+1}_{i,j}$ and $hu^{n+1}_{i,j}$ in which $h^{n+1}_{i,j}$ is needed.

WENO and SSP-ERK

We present a brief overview of the WENO SSP-ERK method used in the solver for the 2D shallow water wave (7.1) that we seek to accelerate using Parareal. Integrating (7.2) over a cell I_{ij} one finds that

$$\begin{aligned}\frac{dQ_{ij}(t)}{dt} = &-\frac{1}{\Delta x \Delta y}\left(\int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} f\left(q\left(x_{i+\frac{1}{2}}, y, t\right)\right) dy + \right. && (7.17)\\ &- \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} f\left(q\left(x_{i-\frac{1}{2}}, y, t\right)\right) dy + \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} g\left(q\left(x, y_{j+\frac{1}{2}}, t\right)\right) dx && (7.18)\\ &\left. - \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} g\left(q\left(x, y_{j-\frac{1}{2}}, t\right)\right) dx + \int_{I_{i,j}} s(q,z)\, d\Omega_I\right) && (7.19)\end{aligned}$$

where $Q_{ij}(t)$ is the cell average as defined in (7.4). We introduce the operator L as an approximation to the RHS of (7.17) by the following conservative scheme

$$L(Q_{ij}) = -\frac{1}{\Delta x}\left(\hat{f}_{i+\frac{1}{2},j} - \hat{f}_{i-\frac{1}{2},j}\right) - \frac{1}{\Delta y}\left(\hat{g}_{i,j+\frac{1}{2}} - \hat{g}_{i,j-\frac{1}{2}}\right) + \frac{1}{\Delta x \Delta y}\int_{I_{i,j}} s(q,z)\; dxdy \tag{7.20}$$

where the numerical flux $\hat{f}_{i+\frac{1}{2},j}$ is defined as

$$\hat{f}_{i+\frac{1}{2},j} = \sum_{\alpha} w_{\alpha} F\left(q^{-}_{i+\frac{1}{2},y_j+\beta_\alpha \Delta y}, q^{+}_{i+\frac{1}{2},y_j+\beta_\alpha \Delta y}\right) \tag{7.21}$$

Here β_α and w_α are Gaussian quadrature nodes and weights for approximating the integration in y as

$$\hat{f}_{i+\frac{1}{2},j} \approx \frac{1}{\Delta y} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} f\left(q\left(x_{i+\frac{1}{2}}, y, t\right)\right) dy \tag{7.22}$$

and $q^{\pm}_{i+\frac{1}{2},y}$ are the WENO reconstructed values, computed as described in [279]. F is the numerical flux as defined in (7.6). The approximation of $\hat{g}_{i+\frac{1}{2},j}$ is defined in the same way, but using (7.7). With $L\left(Q_{ij}\right)$ as defined in (7.20), we perform the integration using the optimal third order SSP Explicit Runge-Kutta scheme

$$\begin{aligned} Q^{(1)} &= Q^n + \Delta t L\left(Q^n\right) \\ Q^{(2)} &= \frac{3}{4} Q^n + \frac{1}{4} Q^{(1)} + \frac{1}{4} \Delta t L\left(Q^{(1)}\right) \\ Q^{n+1} &= \frac{1}{3} Q^n + \frac{2}{3} Q^{(1)} + \frac{2}{3} \Delta t L\left(Q^{(2)}\right) \end{aligned} \tag{7.23}$$

A complete introduction to the WENO family of numerical schemes is outside the scope of this chapter. We refer to [279] for an introduction to ENO and WENO based methods, and [315] for higher order WENO applied to the shallow water wave equation with wetting and drying of cells.

7.2.2 Parareal Convergence

To use Parareal, a “coarse” operator $\mathcal{G}_{\Delta T}$ is required that acts as a preconditioner in the Parareal iteration. In our initial investigations on the 1D equation we found that the choice of the coarse operator has a substantial impact on the speed of convergence of the algorithm. Using Lax-Wendroff or MacCormack schemes, we observed that Parareal initially diverges and only converge at the $k = N$’th iteration. The best result was achieved using Roe’s method introduced in the previous section. In this case we do not observe any instabilities, even for problems where shocks appear.

In a recent paper it has been shown trough a discrete dispersion analysis of Parareal, applied to a linear system of ODEs, that the source of instability is different discrete phase speeds of the coarse and fine level, and that the instability is particularly pronounced for higher wave modes[259]. This matches our observations and may well explain the surprising result that Parareal converges when using this particular combination of fine and coarse solver despite solving a hyperbolic non-linear system of

PDEs. The convergence properties of Parareal on WENO SSP-ERK when solving the shallow water wave equation when using Roe's method as a preconditoner is tested on the 1D problem. We run the numerical experiments for two different resolutions $dx = 0.01$ and $dx = 0.001$, two different time-domain lengths, $T = 0.1$ and $T = 1.0$, and two different decompositions in time $N = 20$ and $N = 50$. For each case, we present the error as a function of k, measured with respect to a reference solution computed on a high-resolution mesh

$$\epsilon_u = \int_\Omega \left| U_{T_N}^k(x) - u(x, T_N) \right| dx \tag{7.24}$$

as well as with respect to the "fine" solution, defined as the solution computed sequentially using the 3rd order accurate WENO SSP-ERK scheme

$$\epsilon_{\mathcal{F}} = \int_\Omega \left| U_{T_N}^k(x) - \mathcal{F}_{\Delta T}^N u(x, T = 0) \right| dx. \tag{7.25}$$

For the numerical experiment we let $g = 1$ and use the initial condition

$$h(x, T = 0) = \begin{cases} 1 - \frac{1}{4}\sin(8\pi x) & \text{if } x < \frac{4}{5} \\ \frac{3}{2} & \text{if } x \geq \frac{4}{5} \end{cases}, \quad hu(x, T = 0) = 0 \tag{7.26}$$

as an initial condition at $T = 0$. The initial condition contains both smooth regions and discontinuities that develop into shocks over time. The water and flow profile initial condition is depicted in Figure 7.1 along with the reference solution at $T = 0.1$ and at $T = 1.0$. The result of the numerical experiment is presented in Figure 7.2 for $N = 20$ and in Figure 7.3 for $N = 50$. From the convergence measurements we make several observations.

- The algorithm converges faster on shorter time-subdomains, in particular when measured with respect to the reference solution. On the coarse mesh with $dx = 0.01$ and time-interval $T_{end} = 0.1$, the Parareal solution reaches the accuracy of the fine solution $\epsilon_{\mathcal{F}} \ll \epsilon_u$, already at the second correction.

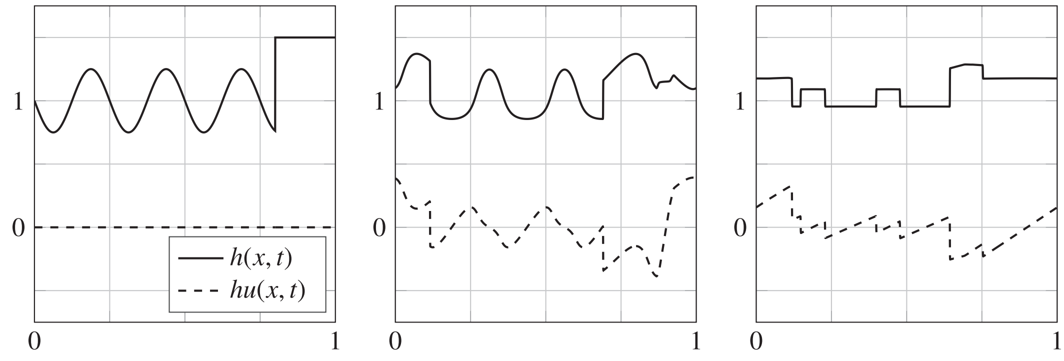
- When the error of the Parareal solution is measured with respect to the WENO solution, the number of iterations needed for convergence to machine accuracy depends only weakly on the time-subdomain length. For the fine mesh, decreasing the time-subdomain has less effect than on the coarse mesh.

- The convergence rate seems to decrease as the mesh is refined, yet is still reasonable even for the very fine mesh using 1000 cells, $dx = 0.001$. Here the accuracy of the Parareal solution is of the same order as that of the WENO fine solution after 4-5 iterations using 20 time-subdomains and 5-7 iterations using 50 time-subdomains.

- Comparing Figures 7.2 and 7.3, it appears that the number of iterations needed before $\epsilon_{\mathcal{F}} \ll \epsilon_u$ is almost unaffected by the total number of time-subdomains. A positive and somewhat surprising result.

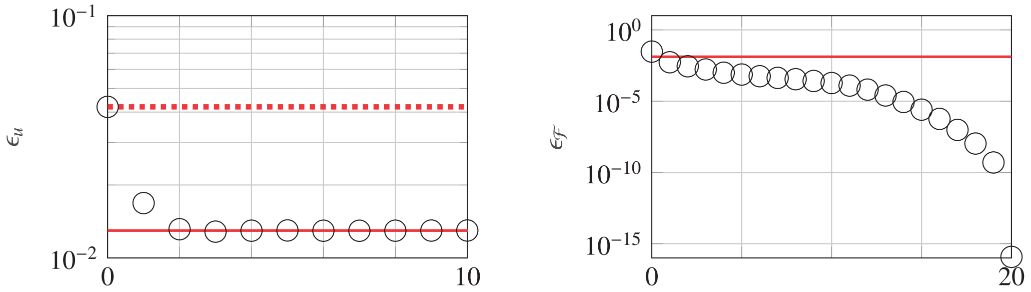
The observations are similar to some of the observations reported in a recent presentation at the 7th Parallel-in-Time workshop[41]. In addition to having the advantage of not introducing dispersive errors with respect to the WENO SSP-ERK solver, the lower order approximate Rieman solver has the advantage of being applied on the same spacial mesh this avoids the use of interpolation between meshes which has been reported as something that can potentially exacerbate any instability issues.

In conclusion, it appears that this coarse operator is suitable for this particular PDE and WENO SSP-ERK discretization. A drawback of the chosen coarse operator is that it is comparatively expensive. It is faster with respect to the fine solver only by a factor 10 to 15 depending on the problem size. In addition, time-subdomains must be short for the algorithm to converge to the accuracy of the WENO SSP-ERK solution quickly. In section 7.3 we introduce a new scheduler for the specific purpose of minimizing the time-subdomain length without making communication of time-subdomains too costly whilst also adaptively overlapping consecutive cycles of Parareal so to mitigate the impact of a relative slow coarse operator.

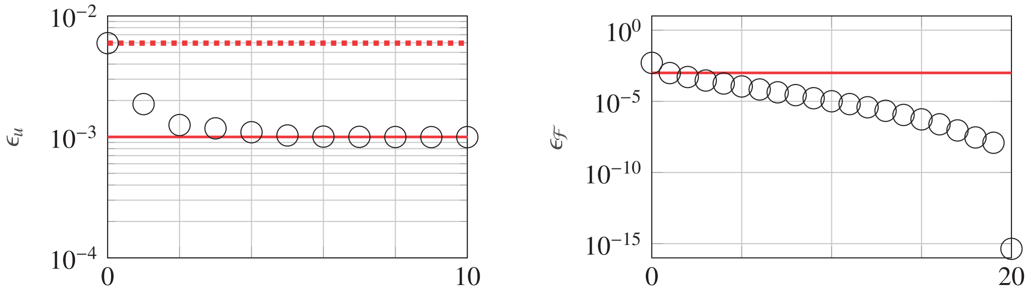


(a) $T = 0$ (b) $T = 0.1$ (c) $T = 1$

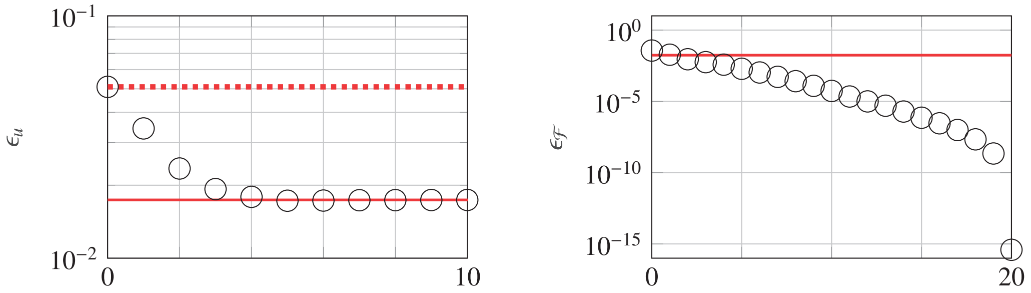
Figure 7.1 – Initial condition (7.26) with periodic boundary conditions. The reference solution is computed using a cell size $dx = 10^{-4}$. (a) $T = 0$. (b) $T = 0.1$. (c) $T = 1$.



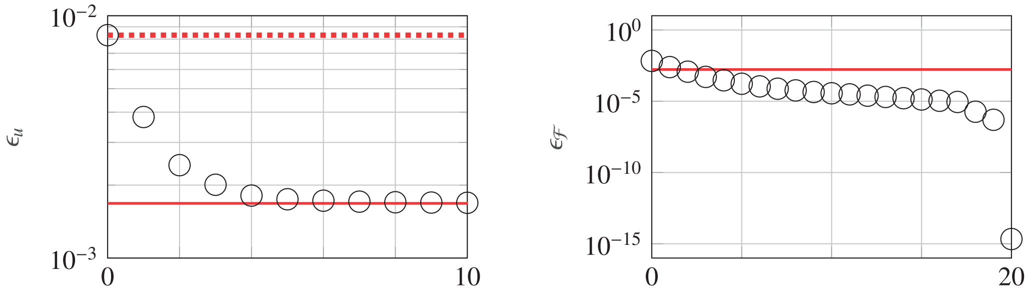
(a) $T_{end} = 0.1, dx = 0.01$ (b) $T_{end} = 0.1, dx = 0.01$



(c) $T_{end} = 0.1, dx = 0.001$ (d) $T_{end} = 0.1, dx = 0.001$

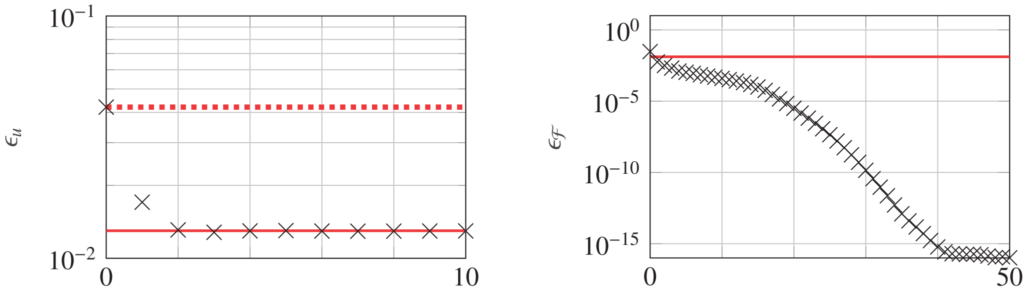


(e) $T_{end} = 1, dx = 0.01$ (f) $T_{end} = 1, dx = 0.01$

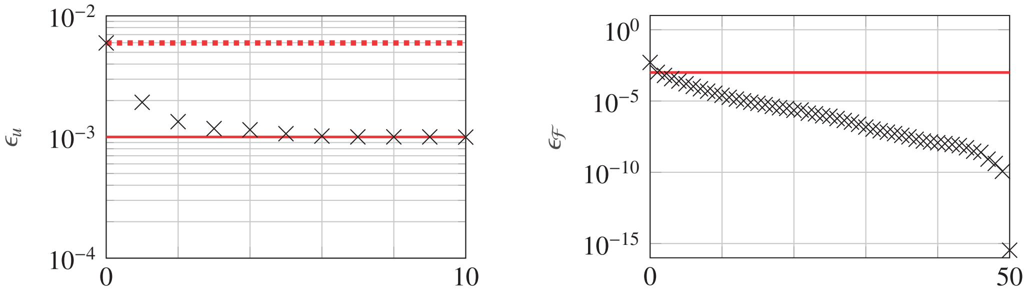


(g) $T_{end} = 1, dx = 0.001$ (h) $T_{end} = 1, dx = 0.001$

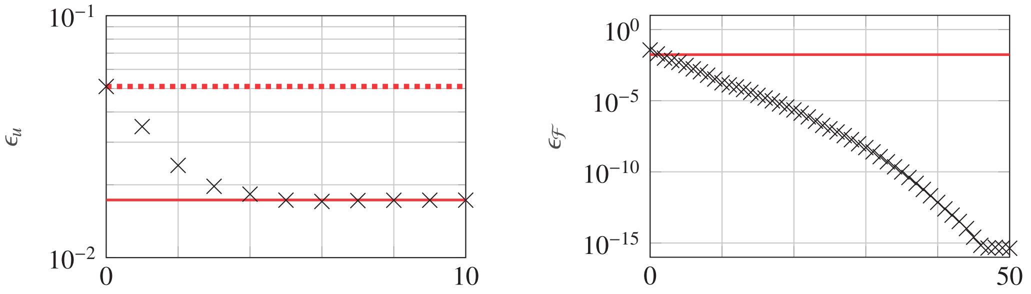
Figure 7.2 – ϵ_u and $\epsilon_{\mathcal{F}}$ as a function of k for the Parareal algorithm when solving the 1D non-linear shallow water wave equation on $N = 20$ time-subdomains using the fine scheme and Parareal preconditioner, described in section 7.2.1. $\epsilon_W u$ and $\epsilon_{\mathcal{F}}$ are defined in (7.24) and (7.25). The red line indicates the error of the fine solution with respect to the true solution. The red dotted line indicates the error of the preconditioner with respect to the true solution.



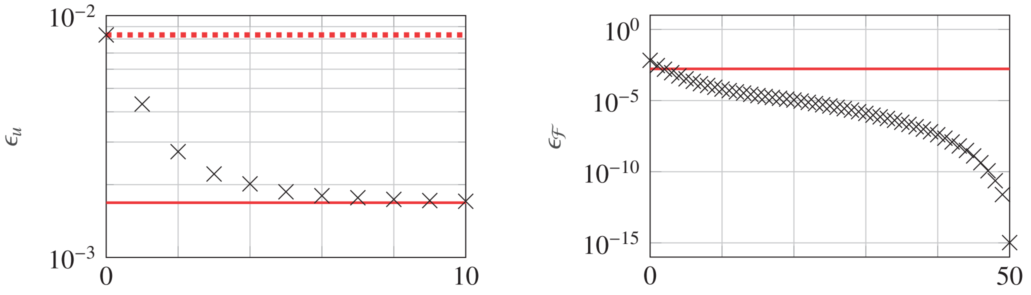
(a) $T_{end} = 0.1, dx = 0.01$ (b) $T_{end} = 0.1, dx = 0.01$



(c) $T_{end} = 0.1, dx = 0.001$ (d) $T_{end} = 0.1, dx = 0.001$

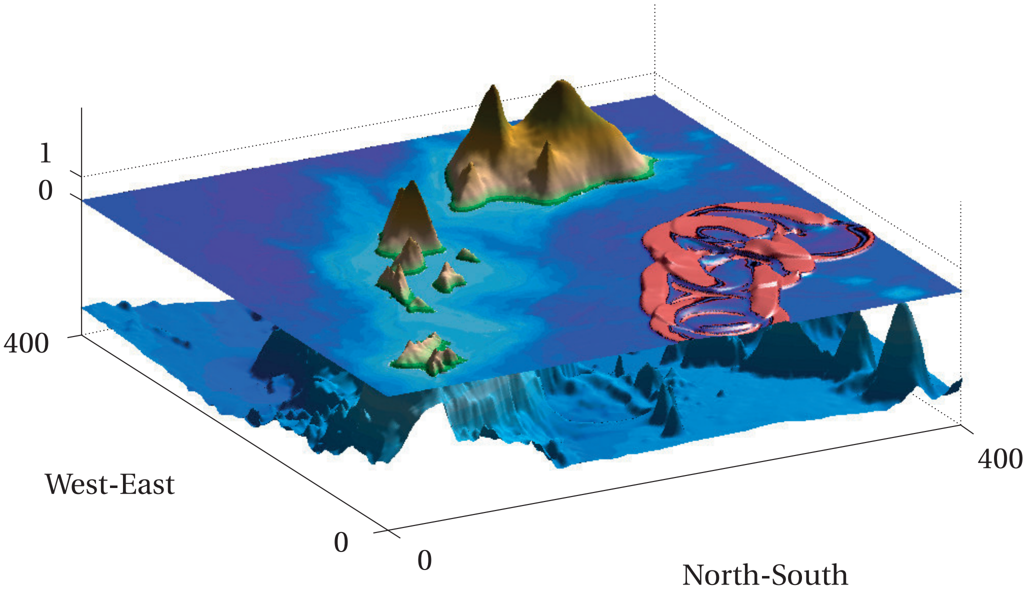


(e) $T_{end} = 1, dx = 0.01$ (f) $T_{end} = 1, dx = 0.01$

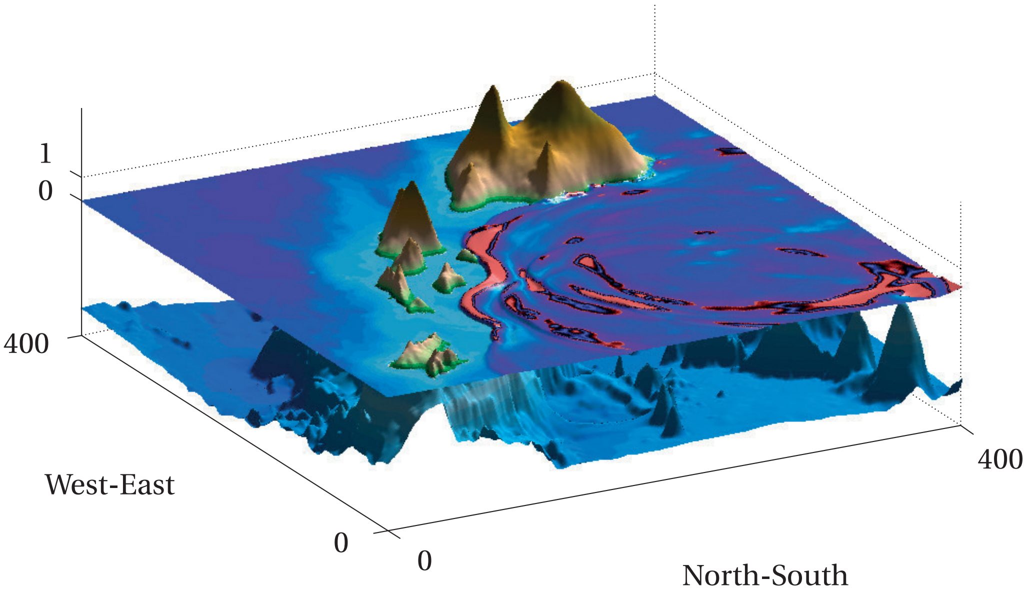


(g) $T_{end} = 1, dx = 0.001$ (h) $T_{end} = 1, dx = 0.001$

Figure 7.3 – ϵ_u and $\epsilon_{\mathcal{F}}$ as a function of k for the Parareal algorithm when solving the 1D non-linear shallow water wave equation on $N = 50$ time-subdomains using the fine scheme and Parareal preconditioner described in section 7.2.1. ϵ_u and $\epsilon_{\mathcal{F}}$ are defined in (7.24) and (7.25). The red line indicates the error of the fine solution with respect to the true solution. The red dotted line indicated the error of the preconditioner with respect to the true solution.



(a)



(b)

Figure 7.4 – Solving the inhomogeneous two-dimensional shallow water wave equation (7.1) with absorbing boundary conditions and a thin-layer mesh reduction technique for land inundation. The equation is solved on a structured rectangular mesh spanning a 400km by 400km map centered over the O'Ahu and Hawaii islands in the pacific. The bathymetry and topography was taken from the free ETOPO1 global relief data-set[4]. The initial conditions at $T = T_0$ consists of multiple superimposed waves rising above the steady-state solution. A red-black-purple coloring scheme is used to indicate the deviation of water height with respect to steady state water-line over time.

7.2.3 Introducing the Test Case

The solution presented in Fig. 7.6 and 7.7 to the test-case was computed using the WENO SSP-ERK scheme.

In this section we present the test-case used to evaluate the extent to which the original Parareal algorithm and CAAP, yet to be presented, can be used to accelerate the process of finding a numerical solution to (7.1). The test-case uses the radially-symmetric elliptic paraboloid bathemetry as a classic test-case presented in [297], but with a different initial condition to allow shocks to develop as time progresses. In the model, simulation of land inundation is included.

The initial condition in the model, proposed by Thacker[297], is a standing half-wave in a radially-symmetric elliptic paraboloid bathymetry. For the test case, Thacker derives an analytical solution. The case is excellent for testing correctness of an implementation, but is insufficient in the context of time-parallel integration as the test-case contains no shocks. In the numerical solution of hyperbolic systems of partial differential equations, an important aspect is the computational challenges associated with handling shocks. It is reasonable to assume that the presence of shocks may have an effect on the convergence rate of the Parareal method. Furthermore the complexity of the solution is limited. Using the simple test-case of [297] may thus lead to a false positive in the sense that observing a fair convergence rate on this particular problem may not say much about the case for hyperbolic problems in general.

Due to these limitations of the classical test-case we instead propose a new shock-containing test-case which is better suited to investigate the extend to which Parareal is applicable for such a problem. We maintain the usage of a radially-symmetric elliptic paraboloid to describe the bottom bathymetry of the basin, given by

$$z\left(x,y\right) = h_0 \frac{r^2}{a^2} \tag{7.27}$$

with $r = \sqrt{\left(x - L/2\right)^2 + \left(y - L/2\right)^2}$ for $(x,y) \in [0,L] \times [0,L]$. Here a is the radius of the basin and h_0 the basin depth at the center of the paraboloid. We define the perturbation h_p to the water surface at rest as

$$h_p\left(x,y\right) = A\cos^2\left(\omega\theta\right)\exp\left(-\frac{\left(r-R\right)^2}{2\sigma^2}\right) \tag{7.28}$$

with $\theta = \arctan\left(\frac{y-.5L}{x-.5L}\right)$. The initial water height may then be written as

$$h\left(x,y,0\right) = \begin{cases} h_0 + h_p\left(x,y\right) - z\left(x,y\right) & \text{if}\, r\left(x,y\right) \leq a \\ 0 & \text{otherwise} \end{cases} \tag{7.29}$$

with the discharges $hu\left(x,y,0\right)=0$ and $hv\left(x,y,0\right)=0$ for all $(x,y)\in[0,L]\times[0,L]$. For the parallel integration tests in Section 7.5, we let $L=1000km$ with a basin radius of $a=400km$ and a depth $h_0=1km$. The peak of the waves H_p at radius $R=300km$ from the center of the map, with an amplitude $A=500m$, frequency $\omega=4$ and width $\sigma=10km$.

In Figure 7.5 the initial condition is depicted on a 1600x1600 cells map. In Figure 7.6 and 7.7 the solution, as computed by the WENO scheme, is presented in 10 minute intervals. The solution contains rich interactions between smooth regions and shocks as well as wetting and drying. The complexity of the solution may be increased by increasing ω. Throughout the remainder of the chapter we keep $\omega=4$ as in the figures.

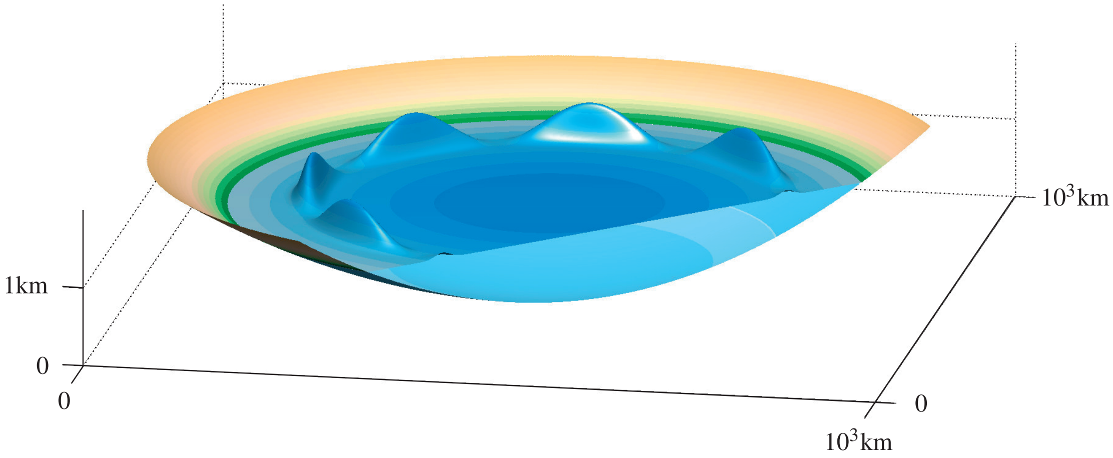
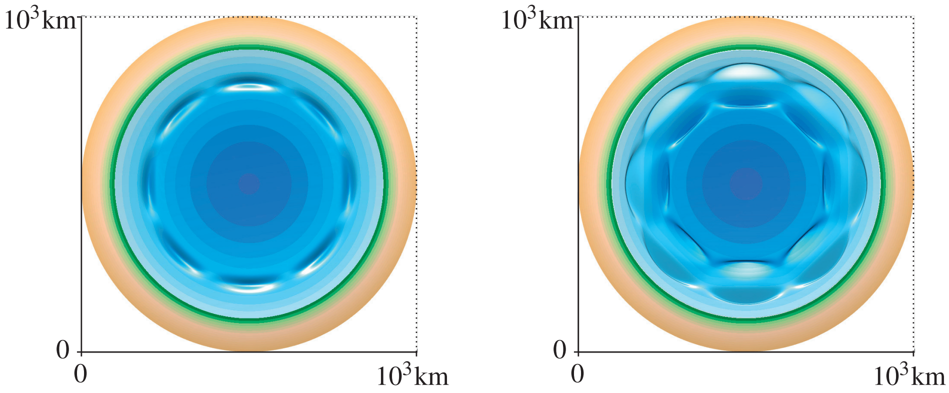
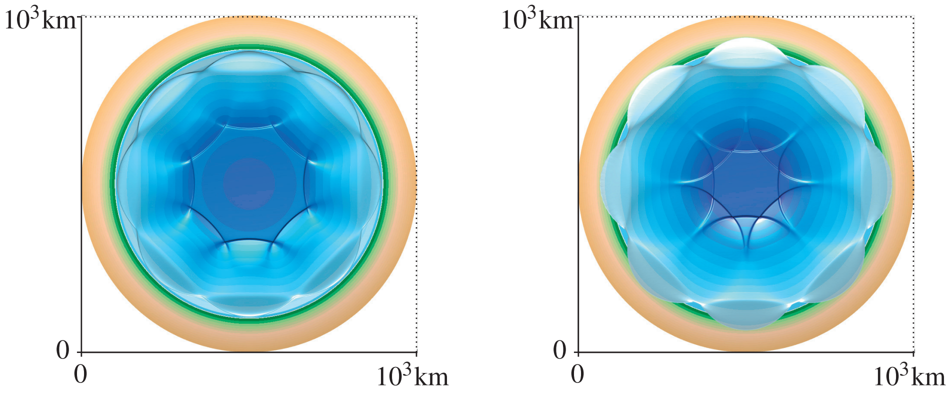


Figure 7.5 – The initial condition, (7.29) for a wave amplitude of $A=500m$, a frequency $\omega=4$ and width a $\sigma=10km$ with the bathymetry (7.27) using $L=1000km$, a basin radius of $a=400km$, and a depth $h_0=1km$. The solution of (7.1) with the initial condition depicted, is given for 10 minute intervals in Fig. 7.6 and 7.7.



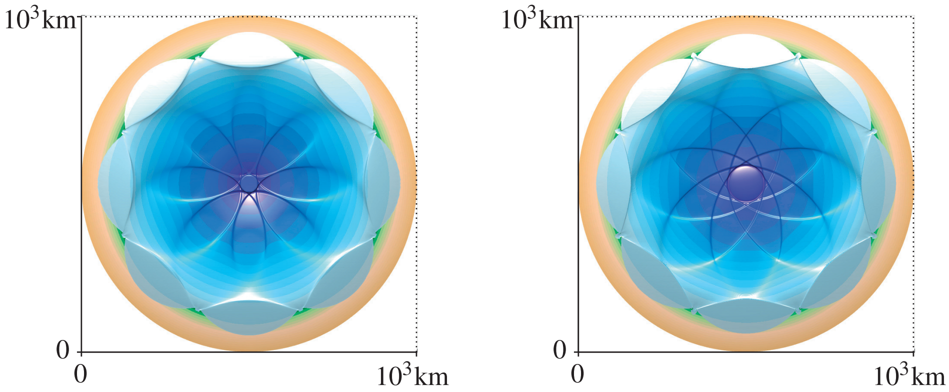
(a) T=0min

(b) T=10min



(c) T=20min

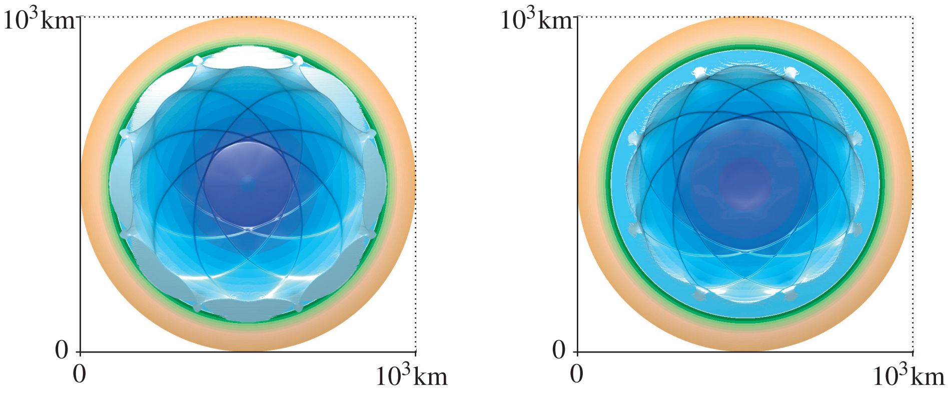
(d) T=30min



(e) T=40min

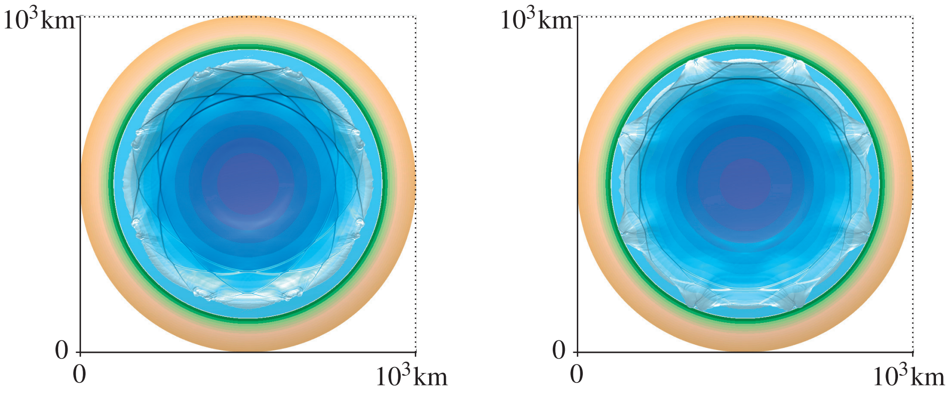
(f) T=50min

Figure 7.6 – Water height as a function of time for the test case (7.29) at 10 minute intervals from $T_0 = 0$ to $T = 50$min. Beige and green colors are used to indicate land whilst shades of blue indicate water depth. Light effects have been added to highlight the location of shocks.



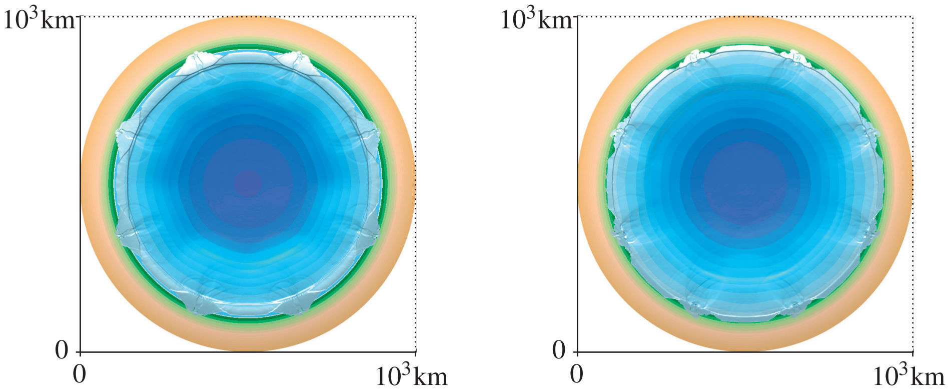
(a) T=60min

(b) T=70min



(c) T=80min

(d) T=90min



(e) T=100min

(f) T=110min

Figure 7.7 – Water height as a function of time for the test case (7.29) at 10 minute intervals from $T = 60$ to $T = 110$min. Beige and green colors are used to indicate land whilst shades of blue indicate water depth. Light effects have been added to highlight the location of shocks.

7.3 Space-Time Domain Decomposition

The simulation to be accelerated uses an explicit numerical scheme, as described in Section 7.2.1, and no linear systems needs to be solved so its parallel-in-space implementation is straightforward. Following each time-step of the explicit Runge-Kutta scheme, a two-cell wide halo is exchanged between all adjacent spatial subdomains using MPI with one rank per subdomain. The parallel-in-space implementation scales well up to 5 nodes (80 cores) on the small 1600x1600 cell test-case using the EPFL Bellatrix cluser. In figure 7.9, the space-time parallel implementation is conceptually depicted. White lines indicate the division of the spatial domains, and each image indicates a time-subdomain.

With 6 nodes and above, the cost of communication between subdomains becomes too large for further parallel acceleration as illustrated in the profiling measurements in Figure 7.8. For further parallel acceleration, we turn to time-parallel techniques. The efficient implementation of the standard Parareal algorithm for combined space and time parallelism is more involved. For Parareal, a coarse operator acting as a preconditioner in the solution of the system (5.9) is needed, and this preconditioner must be parallel in space as well. Unlike the 3rd order WENO SPP-ERK used to solve (7.1), the preconditioner $\mathcal{G}_{\Delta T}$, only requires a single cell halo to be exchanged between the spatial subdomains.

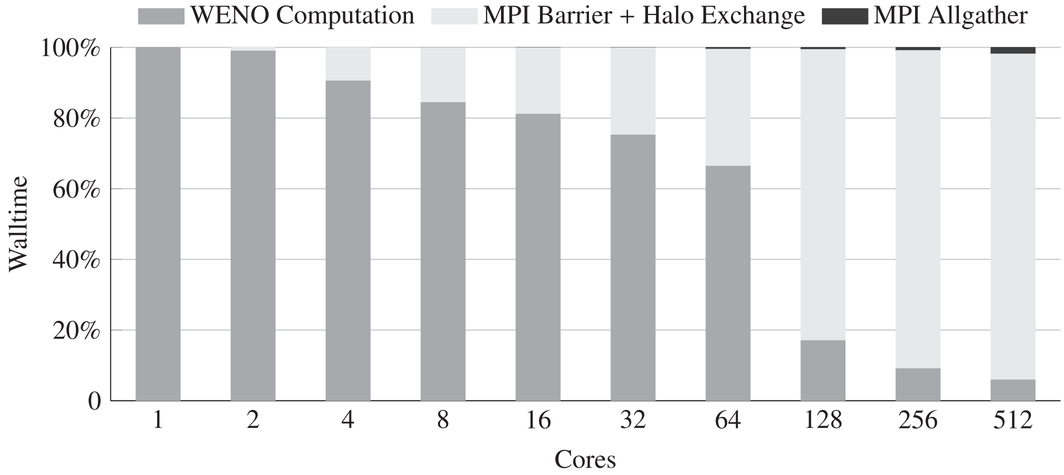


Figure 7.8 – Profiling the parallel-in-space WENO based Tsunami simulation when solving the 1600x1600 cells test-case presented in Section 7.2.3. As the number of cores increase, the cost of the halo-exchange between subdomains comes to dominate. Due to the small size of the test-case, the code effectively stops scaling with 128 cores. The code was profiled on the EPFL Bellatrix cluster. Each node in the cluster contains two 8-core Intel Xeon E5-2660 CPUs and Inifiniband QDR 2:1 connectivity between nodes. Using a single core on a single node, the computation takes 11817 seconds to complete. Using 4 nodes for a total of 64 cores, the computation needs 278 seconds to complete.

Whilst the coarse operator $\mathcal{G}_{\Delta T}$ is trivial to make parallel, efficiently solving the recursive Parareal formulation (5.10) is not so. A number of schedulers for dividing the computational work on clusters have been proposed in the literature. A direct approach for distributing the work is to apply the coarse and fine operators in strictly separate phases, i.e., in each iteration, a number of worker node-groups each compute the application of $\mathcal{F}_{\Delta T}$ in parallel. When completed, data is collected on a manager node that performs the sequential application of $\mathcal{G}_{\Delta T}$, followed by the Parareal corrections, before distributing the data to the worker node-groups for a new iteration. If $\mathcal{G}_{\Delta T}$ is computationally very cheap, the manager-worker approach may work sufficiently well. In practice however, it has been observed repeatedly that this approach is too restrictive for Parareal to achieve speed-up for anything but simple problems. In [9], a better scheduler was introduced, simple in design, yet near optimal in terms of exploiting the dependencies that exists in the recursive tree that defines Parareal. A schematic depiction of the procedure is given in Figure 7.10. The algorithm introduced is equivalent to first executing Algorithm 1 followed by a single execution of Algorithm 2, each of which will be described later.

Parallel-in-time integration with long time-subdomains has certain advantages when using an appropriate scheduler. In this case the communication pattern becomes dominated by a few large time-subdomain interfaces that must be communicated between node-groups. The potential of this latency tolerant communication pattern was investigated in early papers[287, 314].

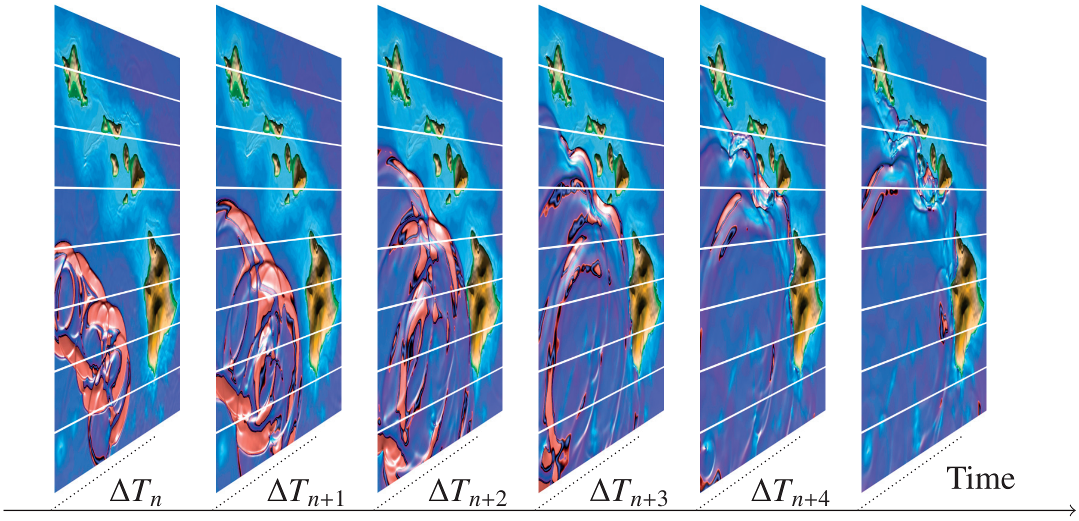


Figure 7.9 – Domain decomposition in space and time of the WENO based solver introduced in Section 7.2. White lines indicate the division of the spatial domains, and each image indicate a subdomain spanning ΔT in time.

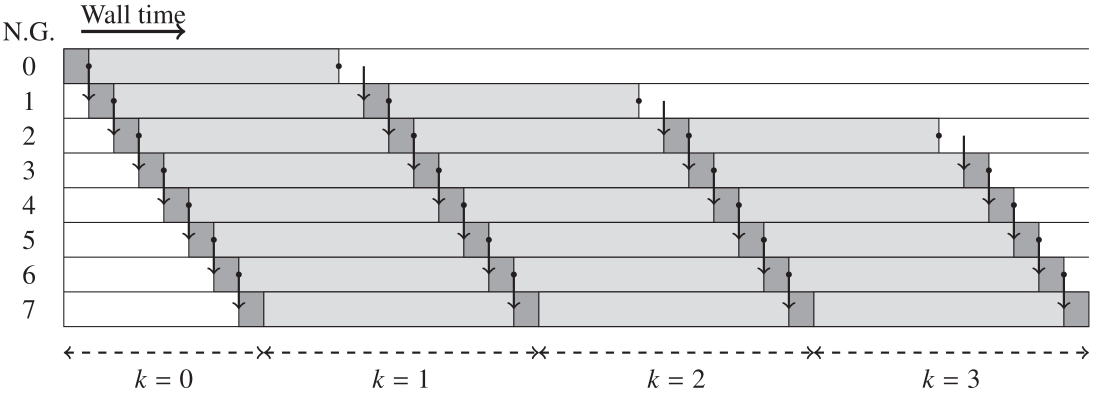


Figure 7.10 – The standard fully distributed Parareal[9]. Each time-subdomain is handled by a unique node-group, possibly parallel-in-space. Dark gray indicates that a node-group is computing the preconditioner $\mathcal{G}_{\Delta T}$, light gray indicates that a node-group is computing $\mathcal{F}_{\Delta T}$. Drawn for $n_t = 8$. Shorter time-subdomains may lead to faster convergence, but this comes at the cost of more frequent communication of the solution-states at time-subdomain interfaces. Drawn as if the tolerance was satisfied by the 3rd correction.

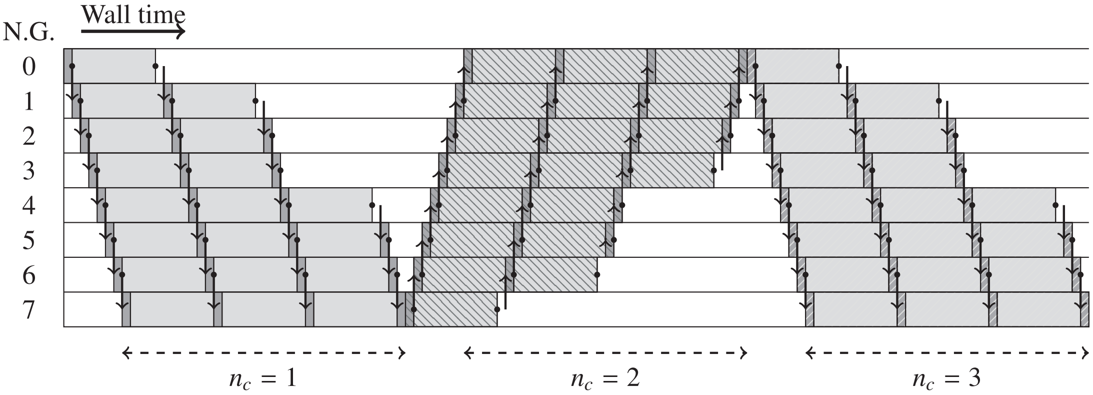
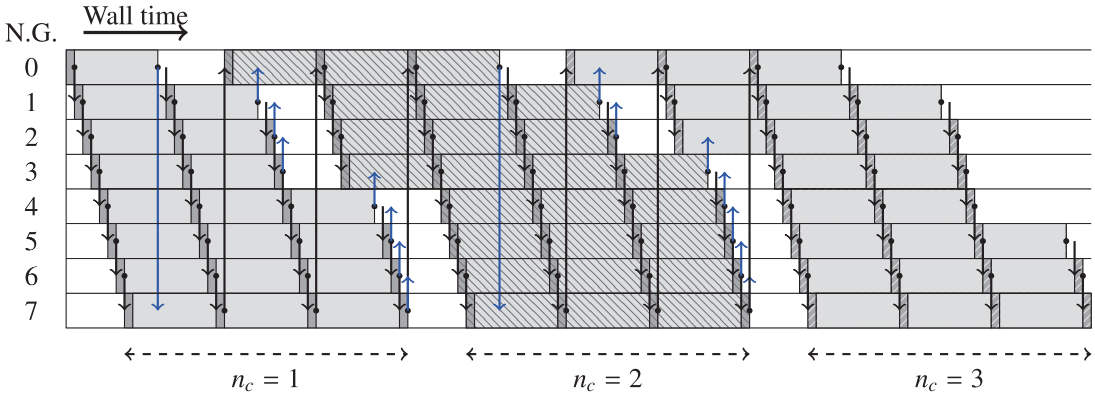
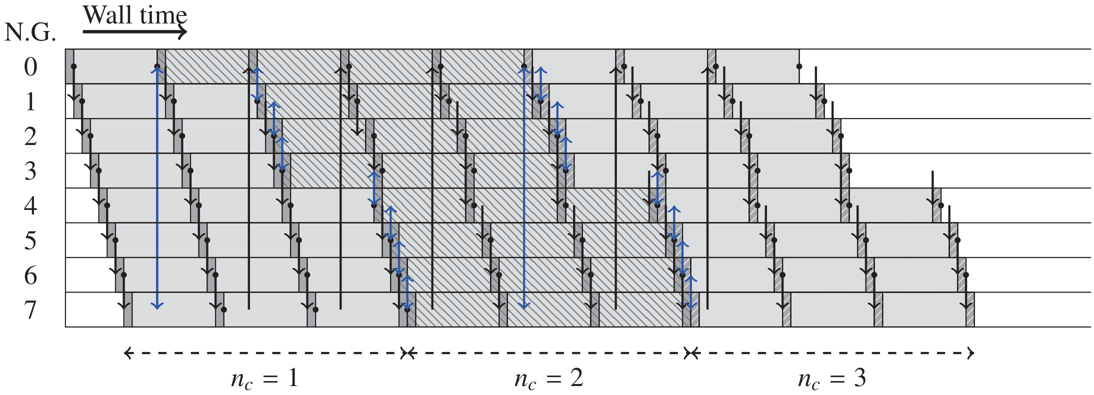


Figure 7.11 – Multiple consecutive executions of the standard fully distributed Parareal[9]. Each time-subdomain is handled by a unique node-group, possibly parallel-in-space. Dark gray indicates that a node-group is computing the preconditioner $\mathcal{G}_{\Delta T}$, light gray indicates that a node-group is computing $\mathcal{F}_{\Delta T}$. Drawn for $n_t = 8$, $n_c = 3$. Shorter time-subdomains may lead to faster convergence, but this comes at the cost of more frequent communication of the solution-states at time-subdomain interfaces. In each cycle, the tolerance is satisfied by the 3rd correction. The direction of information flow shifts with each cycle to reduce the number of time-subdomain interface states sent.



(a) Adaptive scheduler with $\beta = 0$



(b) Adaptive scheduler with $\beta = 1$

Figure 7.12 – Schematic representation of a proposed Adaptive Parareal scheduler. The scheduler lets multiple cycles of Parareal overlap during execution. Drawn here for $n_t = 8, n_c = 3$. Dark gray indicates that a node-group is computing the preconditioner $\mathcal{G}_{\Delta T}$, light gray indicates that a node-group is computing $\mathcal{F}_{\Delta T}$. Black dots and arrows indicate sending and receiving of time-subdomain interface solution state. Blue arrows indicate a signal being sent to inform a node-group working on a time-subdomain, that the next time-subdomain has become active. Each node-group has a boolean flag that indicates if the next time-subdomain is active or not. (a) $\beta = 0$ for a fully patient model in which node-groups that finished a time-subdomain in a cycle will wait for a correction to be made before receiving a new state to commence their work. (b) $\beta = 1$ for a fully impatient model in which node-groups that finished a time-subdomain in a cycle will receive a new state to commence their work immediately.

Convergence of Parareal is slow for most problems on long intervals as has been demonstrated in many early papers, see [228] for an overview, and later established rigorously in [126].

To achieve faster convergence one must therefore yet again divide the interval to be integrated into smaller intervals in which we can apply multiple cycles of Parareal. This could be done as outlined in Figure 7.11 with Algorithm 3, using multiple consecutive cycles of Parareal, implemented with the scheduler proposed in [9]. In [38], an "event-based" Parareal scheduler was proposed. Here a data-dependency driven approach is taken to scheduling i.e. when all dependencies are satisfied for a time-subdomain, the work is scheduled to an available node.

In the scheduler we present here, we combine the simplicity of the scheduler introduced in [9], with the scalability of [38], whilst introducing a parameter to tune the trade-off between node-group occupancy and speed of convergence. We denote this proposed work scheduler as Adaptive Parareal. The fundamental approach is to let adjacent cycles of Parareal overlap in execution time. When a node-group completes its work on a time-subdomain in a given cycle, it will immediately begin working on the closest inactive time-subdomain in the next cycle.

This may happen in one of two ways: The node-group may commence working on the time-subdomain on the next cycle using the most recent iteration available on the preceding time-subdomain. Alternatively it may wait for the next iteration on the preceding time-subdomain to become available. Which choice is better is not obvious and will be situation and problem dependent. If a recent iterate is soon to be available, it may be better waiting. If, on the other hand, the preceding time-subdomain has just commenced work on a new iteration, it may be better to initiate work on the most recent iteration available rather than waiting for the preceding time-subdomain to complete.

We introduce a parameter $\beta \in [0,1]$ that controls how patient the node-groups shall be. When a node-group receives a signal that the next time-subdomain has become active, it will immediately return a solution state if the progress on the application of $\mathcal{F}_{\Delta T}$ is less than β. The progress indicator on $\mathcal{F}_{\Delta T}$ could for example be on the time that has been integrated relative to the total time-subdomain length. Thus, if β is small, the scheduler is patient and if β is large the scheduler is impatient, and will value minimizing idle nodes over convergence in a small number of iterations.

The communication pattern in this model is asynchronous. The iteration and time at which the convergence criteria is satisfied, is unknown for all time-subdomains. Hence it is not possible a priori to predict the communication pattern. To enable node-groups to signal their status, and potentially send a time-subdomain interface, while they are in the process of computing $\mathcal{F}_{\Delta T}$, a separate signal thread is needed.

Algorithm 1 Initialization procedure for one or more Parareal cycles. The algorithm is used repeatedly in Algorithm 3 when a new cycle needs to be initiated, and also used in the adaptive variant for the workthread Algorithm 5 to initiate the 0th iteration on the 1st cycle. Running the initialization once followed by Algorithm 2 once will produce an execution pattern as schematically depicted in Figure 7.10.

```
    Input $\mathbf{U}^k_{id_{\Delta T}-1}$, $FirstNodeGroup$, $LastNodeGroup$, $tag\_send$, $Converge$, $ConvergeNext$
    Output $\tilde{\mathbf{U}}^0_{id_{\Delta T}}$, $\mathbf{U}^0_{id_{\Delta T}}$, $tag\_send$, $Converge$, $ConvergeNext$
$k \leftarrow 0$
if $FirstNodeGroup$ then
    $\tilde{\mathbf{U}}^0_{id_{\Delta T}} \leftarrow \mathcal{G}_{\Delta T}\mathbf{U}^k_{id_{\Delta T}-1}$
    $\mathbf{U}^0_{id_{\Delta T}} \leftarrow \tilde{\mathbf{U}}^0_{id_{\Delta T}}$
    Send $Converge$ and $\mathbf{U}^0_{id_{\Delta T}}$ on forw_intercomm
    $tag\_send = tag\_send + 1$;
    $ConvergeNext \leftarrow$ TRUE
else
    Recv $Converge$ and $\mathbf{U}^0_{id_{\Delta T}-1}$ on back_intercomm
    $\tilde{\mathbf{U}}^0_{id_{\Delta T}} \leftarrow \mathcal{G}_{\Delta T}\mathbf{U}^0_{id_{\Delta T}-1}$
    $\mathbf{U}^0_{id_{\Delta T}} \leftarrow \tilde{\mathbf{U}}^0_{id_{\Delta T}}$
    if $!LastNodeGroup$ then
        Send $Converge$ and $\mathbf{U}^0_{id_{\Delta T}}$ on forw_intercomm
        $tag\_send = tag\_send + 1$;
    end if
end if
```

Each signal thread will receive $n_c - 1$ signals, and each time a signal is received it indicates that next time-subdomain has become active. When receiving a signal, it sets the status of the flag, indicating if the next time-subdomain is active, to true. After doing so, the thread checks if the progress indicator on $\mathcal{F}_{\Delta T}$ is smaller than β. If it is, it will send the most recent iterate of the time-subdomain interface that it is computing. In our implementation, Posix Threads was used to create the signal threads needed in the adaptive scheduler.

Schematic examples for $\beta = 0$ and $\beta = 1$ are presented in Figure 7.12a and 7.12b, respectively. In these figures, black dots and arrows indicate the sending and receiving of time-subdomain interfaces. The blue arrows indicate the signal that a time-subdomain has become active. Pseudo code for the proposed adaptive scheduler is given in Algorithm 5 along with Algorithm 4 for the corresponding signal thread. Separate communicators are created for each spatial-subdomain, and all communication of time-subdomain interfaces happens through dedicated inter-communicators. Doing so provides encapsulation of the application code already written, whilst ensuring a natural distinction between the parallelism in computing derivatives in space and parallelism in the time integration procedure. The asynchronous adaptive scheduler presented is not trivial to implement. To ease implementation and understanding, we have presented the new scheduler as the execution of different procedures that may be implemented and tested independently. Simply executing Algorithm 1, once followed by Algorithm 2 once is equivalent to the "fully distributed" Parareal from [9]. The two procedures form the essential components Algorithm 3 for multiple consecutive cycles of Parareal as well as the proposed adaptive scheduler Algorithm 5 and Algorithm 4.

Algorithm 2 A single Parareal cycle. Identical to the model introduced in [9] when combined with the initialization in Algorithm 1. This algorithm is used in Algorithm 3 for multiple cycles of Parareal, as well as in Algorithm 5 for the adaptive scheduler.

```
Input   $\mathbf{U}^{k-1}_{id_{\Delta T}-1}$, $\tilde{\mathbf{U}}^{k-1}_{id_{\Delta T}}$, $k$, $LastNodeGroup$, $tag\_send$, $Converge$, $ConvergeNext$
Output  $\hat{\mathbf{U}}^{k-1}_{id_{\Delta T}}$, $tag\_send$, $Converge$, $ConvergeNext$
while !$Converge$ do
    $k \leftarrow k + 1$
    $\hat{\mathbf{U}}^{k-1}_{id_{\Delta T}} \leftarrow \mathcal{F}_{\Delta T}\mathbf{U}^{k-1}_{id_{\Delta T}-1}$
    if $ConvergeNext$ then
        $Converge \leftarrow$ TRUE
        $\mathbf{U}^{k}_{id_{\Delta T}} \leftarrow \hat{\mathbf{U}}^{k-1}_{id_{\Delta T}}$
        if !$LastNodeGroup$ then
            Send $Converge$ and $\mathbf{U}^{k}_{id_{\Delta T}}$ on forw_intercomm
            $tag\_send = tag\_send + 1$;
        end if
        break
    end if
    Recv $Converge$ and $\mathbf{U}^{k}_{id_{\Delta T}-1}$ on back_intercomm
    $\tilde{\mathbf{U}}^{k}_{id_{\Delta T}} \leftarrow \mathcal{G}_{\Delta T}\mathbf{U}^{k}_{id_{\Delta T}-1}$
    $\mathbf{U}^{k}_{id_{\Delta T}} \leftarrow \tilde{\mathbf{U}}^{k}_{id_{\Delta T}} + \hat{\mathbf{U}}^{k-1}_{id_{\Delta T}} + \tilde{\mathbf{U}}^{k-1}_{id_{\Delta T}}$
    if $Converge$ & $|\mathbf{U}^{k}_{id_{\Delta T}} - \mathbf{U}^{k-1}_{id_{\Delta T}}| > \epsilon$ then
        $Converge \leftarrow$ FALSE
        $ConvergeNext \leftarrow$ TRUE
    end if
    if !$LastNodeGroup$ then
        Send $Converge$ and $\mathbf{U}^{k}_{id_{\Delta T}}$ on forw_intercomm
        $tag\_send = tag\_send + 1$;
    end if
    if $Converge$ then
        break
    end if
end while
```

Algorithm 3 Pseudocode for a Parareal implementation, running n_c consecutive cycles of Parareal, each with n_t time-subdomains. The direction of information flow shifts with each cycle to reduce the number of time-subdomain interfaces solution states to be sent. Schematical example in Figure 7.11.

```
Input   $id_{ng}$, $u_0$
Output  $\mathbf{U}$
$id_{\Delta T} \leftarrow id_{ng}$
$\mathbf{U}_0^0 \leftarrow u_0$
back_intercomm ← intercomm between node-groups $id_{ng}$ and $id_{ng} - 1$
forw_intercomm ← intercomm between node-groups $id_{ng}$ and $id_{ng} + 1$
for $i = 1$ to $n_c$ do
    $tag\_send$ = $tag\_send$ + 1;
    $id_{\Delta T} \leftarrow (i-1)n_t + id_{ng}$        ▷ Set which node-group $id_{ng}$ computes which time-domain $id_{\Delta T}$
    $converge, convergeNext$ FALSE
    $FirstNodeGroup, LastNodeGroup \leftarrow$ FALSE
    if ( $id_{ng} = 1$ and mod $i = 1$ ) or ( $id_{ng} = n_t$ and mod $i = 0$ ) then
        $FirstNodeGroup \leftarrow$ TRUE
    end if
    if ( $id_{ng} = 1$ and mod $i = 0$ ) or ( $id_{ng} = n_t$ and mod $i = 1$ ) then
        $LastNodeGroup \leftarrow$ TRUE
    end if
    procedure Algorithm 1                        ▷ Initiate the $i$th cycle
    procedure Algorithm 2                        ▷ Do Parareal on the $i$th cycle
    procedure Swap back_intercomm and forw_intercomm
end for
```

Algorithm 4 Signal thread for asynchronous communication in the adaptive Parareal Algorithm 5. The thread posts a blocking recv and waits until the node-group handling time-subdomain $id_{\Delta T} + 1$ sends a signal that it is active. If $k > 0$ **and** $status\,(\mathcal{F}_{\Delta T}) < \beta$ **and** $tag\_send = 0$, the thread sends the current solution state $id_{\Delta T}$.

```
Shared  $\beta$, $k$, $Converge$, $\mathbf{U}$, $LastNodeGroup$, $tag\_send$
for $i = 1$ to $n_c - 1$ do
    Recv $LastNodeGroup$ on forw_intercomm
    if $k > 0$ and $status\,(\mathcal{F}_{\Delta T}) < \beta$ and $tag\_send = 0$ then
        $Converge \leftarrow$ FALSE
        Send $Converge$ and $\mathbf{U}_{id_{\Delta T}}^{k-1}$ on forw_intercomm.
        $tag\_send$ = $tag\_send$ + 1;
    end if
end for
```

Algorithm 5 Pseudocode for an adaptive Parareal implementation with n_c cycles each with n_t simultaneously active time-subdomains. Schematic examples of the scheduler is given in Figure 7.12a for $\beta = 0$ and Figure 7.12b for $\beta = 1$. Pseudo code for the signal thread is given in Algorithm 4.

```
Input  $id_{ng}$, $u_0$, $\beta$
Output  U
procedure: Initiate algorithm 4 on separate thread
back_intercomm ← intercomm between node-groups $id_{ng}$ and $id_{ng} - 1$
forw_intercomm ← intercomm between node-groups $id_{ng}$ and $id_{ng} + 1$
$Converge, ConvergeNext, FirstNodeGroup, LastNodeGroup \leftarrow$ FALSE
$id_{\Delta T} \leftarrow id_{ng}$, $\mathbf{U}_0^0 \leftarrow u_0$, $i \leftarrow 0$
if $id_{ng} = 1$ then
    $FirstNodeGroup \leftarrow$ TRUE
end if
if $id_{ng} = n_t$ then
    $LastNodeGroup \leftarrow$ TRUE
end if
procedure Algorithm 1                                  ▷ 0th iteration of the first cycle.
while $i < n_c$ do
    $id_{\Delta T} \leftarrow i \cdot n_t + id_{ng}$
    procedure Algorithm 2
    $i \leftarrow i + 1$, $k \leftarrow 0$, $tag\_send \leftarrow 0$
    if $i < n_c$ then                                  ▷ Initiate a new cycle
        Send $LastNodeGroup$ on back_intercomm
        $LastNodeGroup \leftarrow$ TRUE,    $ConvergeNext \leftarrow$ FALSE
        Recv $Converge$ and $\mathbf{U}^k_{id_{\Delta T}-1}$ on back_intercomm
        if $Converge$ then ▷ Previous time-subdomain converged, local domain will converge next.
            $Converge \leftarrow$ FALSE,    $ConvergeNext \leftarrow$ TRUE
        end if
        $\tilde{\mathbf{U}}^k_{id_{\Delta T}} \leftarrow \mathcal{G}_{\Delta T} \mathbf{U}^k_{id_{\Delta T}-1}$
        $\mathbf{U}^k_{id_{\Delta T}} \leftarrow \tilde{\mathbf{U}}^k_{id_{\Delta T}}$
        $k \leftarrow 1$
        if !$LastNodeGroup$ and $tag\_send = 0$ then   ▷ If next NG active, send time-subdomain interface state.
            Send $Converge$ and $\mathbf{U}^{k-1}_{id_{\Delta T}}$ on forw_intercomm
            $tag\_send$ = $tag\_send$ + 1;
        end if
    end if
end while
```

7.3.1 CAAP Convergence

The convergence properties of a regular single cycle of the Parareal method have been well studied both theoretically [14, 128, 125, 312] and experimentally by [258, 289]W. The convergence properties of multiple consecutive cycles of Parareal and, in particular, when consecutive cycles are allowed to overlap, as in the proposed adaptive scheduler, might be very different and no theory exists on the topic. To investigate how the adaptive scheduler with overlapping cycles may influence the convergence pattern, we test the scheduler on a small numerical example for which an analytical solution is known, computing an approximation to the scalar complex initial value problem

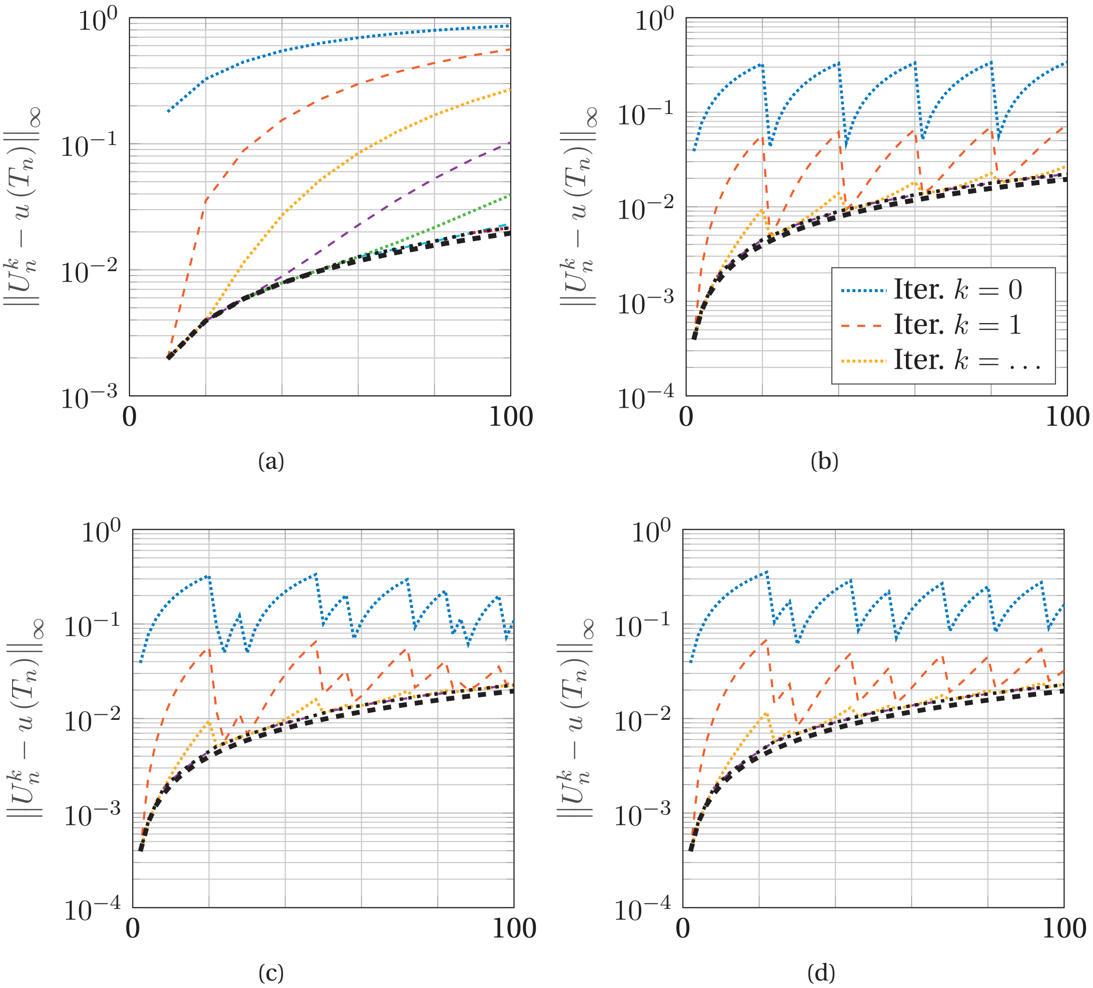
$$\frac{d}{dt}u\left(t\right)=\lambda u\left(t\right), \quad u_0=1,\, \lambda=i \tag{7.30}$$

with $\mathcal{F}_{\Delta T}$, and the preconditioner $\mathcal{G}_{\Delta T}$ given by

$$\mathcal{F}_{\Delta T}U_n^k=\left(1-\lambda dt\right)^{-\frac{\Delta T}{dt}}U_n^k, \quad \mathcal{G}_{\Delta T}U_n^k=\left(1-\lambda dT\right)^{-\frac{\Delta T}{dT}}U_n^k \tag{7.31}$$

using $dt = 10^{-5}$ and $dT = 10^{-3}$ in $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ respectively. In Figure 7.13 convergence for the test equation (7.30) is presented with $n_c = 5$ cycles. Since an analytical solution to (7.30) exists for all times t, we can measure the error as a function of t for each iteration to gain insight into how the Algorithm converges. The convergence criteria is satisfied in substantially fewer iterations using $n_c = 5$ time-subdomains, as can be seen in Figure 7.13b, than when using only a single cycle as in Figure 7.13a. The convergence of the adaptive scheduler with $\beta = 0$ and $\beta = 0.8$ is presented in Figure 7.13c and Figure 7.13d, respectively. The pattern of convergence is clearly different, but the adaptive overlapping of time-subdomains in adjacent cycles appears to work well.

Appendix B contains 8 figures investigating how varying the accuracy of the coarse operator dt, the length of the timedomain ΔT, and the tolerance on the norm between two consecutive iteratives used to determine if a time-subdomain should be accepted as having converged impacts the convergence of both regular Parareal and CAAP on the test equation 7.30. A particularly interesting finding is how it is paramount to choose the tolerance ϵ sufficiently small as otherwise errors will accumulate over . This has happened in Figures B.6(e)-(f) and B.8(e)-(f). Due to the tolerance not being sufficiently low, time-subdomains are being accepted as having converged too early, and over the course of many cycles errors accumulate to the extent that the final solution at $T = T_{end}$ is not of the same level of accuracy as that of the sequential operation. Using a tolerance too large may result in reaching a solution that is not sufficiently accurate, however one does not want to choose the tolerance too low either as this may adversely effect the parallel efficiency. So with CAAP as for Parareal, choosing the tolerance may greatly impact parallel efficiency. How to effectively stop the algorithm is still an open question[60]. The drawings of the work scheduler computation and communication presented in Figures 7.10 and 7.12 are both conceptual, made to present the algorithms. Appendix A contains similar figures, but rather than being purely conceptual, they are made from timings taken, and written to a file, during the parallel solution procedure of solving the ODE 7.30. In the example, the computational work load of $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ is very light, pause functions inside $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ are therefore used to define the computational cost so to demonstrate how the CAAP scheduler works and how it differs from the alternative of simply combing many consecutive cycles of Parareal with a regular scheduler.



(a)

(b)

(c)

(d)

Figure 7.13 – Error of U_n^k with respect to the analytic solution to (7.30) as a function of t for parallel-in-time integration of (7.1) with time-steps and tolerance $\epsilon = 0.01$, used for the measurements in Figure 7.14. (a) Standard Parareal, $n_c = 1$. (b) Standard Parareal, $n_c = 5$. (c) Adaptive Parareal, $n_c = 5$, $\beta = 0$. (d) Adaptive Parareal, $n_c = 5$, $\beta = 0.8$. The black dashed line indicates accuracy of the fine operator with respect to the analytical solution, and the black dotted line indicates the accuracy of $U_n^{k_{conv}}$ obtained at the iteration for which the convergence criteria was satisfied.

7.4 Optimizing Time-Subdomain Length

In the previous section we introduced a scheduler for efficiently executing multiple consecutive Parareal cycles. It does so by letting adjacent cycles overlap in execution time, and by adaptively choosing which previous iteration to initiate a new cycle from. This means that that we are effectively free to choose whatever time-subdomain length we wish, regardless of the number of active subdomains in time to use. Given some IBVP problem, and a fixed number of nodes n_t at our disposal to do parallel-in-time integration of some fixed (long) time-interval $[0, T_{end}]$, the question arises of what time-subdomain length ΔT should one choose?

Since the question is posed for some fixed n_t and T_{end}, it is equivalent to asking: How many cycles n_c of Parareal should we split our time domain of length T_{end} into? The purpose of this section is to develop an approach that allows for an informed choice on ΔT before running the code.

The original Parareal algorithm was presented as parallel-in-time integration of a fixed time interval with the time-subdomain $\Delta T = \frac{T_{end}}{n_t}$. In practice, with this approach, there is a limit to how long T_{end} may be. Therefore, for integration over a long time interval, one must decouple the number of independent time-subdomains from the total time-domain to be integrated. This can be done either trough a simple stop-start strategy of multiple consecutive "cycles" of plain Parareal, as depicted in Figure 7.11 for $n_c = 3$ cycles, or with a more advanced approach were consecutive cycles are allowed to overlap in execution time across nodes, as we introduced with the adaptive scheduler Algorithm 5 and visualized schematically in Figure 7.12.

Decoupling the time-subdomain length ΔT from n_t and T_{end}, in addition to allowing for integration of long time-domains, also introduces the freedom to choose the time-subdomain length as one deem appropriate for achieving high parallel efficiency. When choosing a time-subdomain length, one makes a fundamental trade-off between how often to run the coarse operator and communicate the full solution-state sequentially across all active time subdomains, and the speed at which the algorithm converges. If the time-subdomain ΔT is chosen to be very short, one can expect fast convergence in a few iterations k. However, the algorithm may then fail to provide any parallel acceleration because the preconditioner $\mathcal{G}_{\Delta T}$ will have to be applied more frequently, meaning that the time-subdomain interface solution-states $\mathbf{U}_k^n$ have to be communicated across node-groups more often. Conversely, if one choose a "very long" time-subdomain ΔT, up to $\Delta T \leq \frac{T_{end}}{n_t}$, we may not be limited by the sequential execution of $\mathcal{G}_{\Delta T}$ and communication of solution states $\mathbf{U}_k^n$, but the algorithm may instead need many iterations to convergence, limiting the extent to which parallel acceleration is possible. Ideally, we want to choose ΔT so that these effects are balanced in such a way that we achieve the highest possible speed-up. Finding such a ΔT is made complicated by the fact that we do not know in general how the number of iterations to convergence, k_{conv}, depends on n_t and ΔT. As we shall see, this does however not mean that we must make an uninformed guess when choosing ΔT.

7.4.1 Single Cycle Analysis

Before embarking on an analysis of the multi-cycle Parareal algorithm presented in Section 7.3, we consider the standard "single cycle" Parareal algorithm. Henceforth we refer to the collection of CPU's and possible co-processors that compute the application of $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ to $\mathbf{U}_k^n$ as a node-group. With this notation we abstract away whatever (spatial) domain-decomposition that may have been applied in constructing $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$. The parallel speed-up of a single cycle of length $\delta T = n_t \Delta T = T_{end}$, as presented in Figure 7.11, can be written as

$$\psi = \frac{n_t C_{\mathcal{F}} \Delta T}{n_t \left(C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \right) + \kappa \left(n_t, \Delta T \right) \left(C_{\mathcal{F}} \Delta T + C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \right)} \tag{7.32}$$

The term in the numerator is the time it takes to compute the solution sequentially, the two terms in the denominator measures the time it takes to compute the Parareal solution. The first of these two terms is the time-consumption of computing iteration 0, the second term expresses the combined time-consumption of all subsequent iterations. n_t denotes the number of time-subdomains, i.e. the number of node groups that may be used. We assume that there exists some constants $C_{\mathcal{G}}$ and $C_{\mathcal{F}}$ proportional to ΔT so that

$$T_{\mathcal{F}}^w = C_{\mathcal{F}} \Delta T, \quad T_{\mathcal{G}}^w = C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \tag{7.33}$$

where $T_{\mathcal{F}}^w$ and $T_{\mathcal{G}}^w$ denotes the wallclock-time it takes for the two operators $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ to be applied to a state $\mathbf{U}_k^n$ when computed on a node-group. $T_{\mathcal{C}}^w$ denotes the wallclock-time it takes to communicate a solution state from one node-group to another. $T_{\mathcal{C}}^w$ is included in the complete execution time of the preconditioner $\mathcal{G}_{\Delta T}$ since for every application of $\mathcal{G}_{\Delta T}$ there will be one state $\mathbf{U}_k^n$ that must be communicated from one node-group to another. The function $\kappa : \mathbb{N}^+ \times \mathbb{R}^+ \to \mathbb{N}^+$ is the number of iterations needed to achieve the convergence criteria. It is important to note here that the only unknown is $\kappa \left(n_t, \Delta T \right)$. ΔT and n_t are known and the rest are constants that can be measured for a specific cluster on a given problem. Let us first explore the limit as the time-subdomain length ΔT becomes small. Assuming that $\lim_{\Delta T \to 0} \kappa \left(n_t, \Delta T \right) = 1$, from (7.32) one finds that

$$\lim_{\Delta T \to 0} \psi = \frac{n_t}{\left(n_t + 1 \right)} \frac{T_{\mathcal{F}}^w}{T_{\mathcal{C}}^w} \tag{7.34}$$

becomes an upper bound for achievable speed-up. This is not surprising, as by Amdahl's law, $T_{\mathcal{F}}^w / T_{\mathcal{G}}^w$ must be an upper limit to speed-up for the Parareal algorithm. Equation (7.34) simply states that for small ΔT, the cost $T_{\mathcal{C}}^w$ of communicating the solution states $\mathbf{U}_k^n$ sequentially across nodes becomes the dominating term that limits parallel speedup. Whilst (7.34) may be descriptive, it does not let us choose ΔT in any meaningful way. Our goal is to find the ΔT that maximize (7.32), so let us frame the problem in the form of an optimization problem

$$\Delta T_{opt} \ = \ \underset{\Delta T \in \mathbb{R}^+}{\arg\max}\, \psi\left(\Delta T\right). \tag{7.35}$$

The derivative of $\psi\left(\Delta T\right)$ w.r.t. ΔT is

$$\begin{aligned} \frac{\partial \psi\left(\Delta T\right)}{\partial\left(\Delta T\right)} &= \frac{\partial}{\partial\left(\Delta T\right)} n_t C_{\mathcal{F}} \Delta T \left(n_t\left(C_{\mathcal{G}}\Delta T + T_{\mathcal{C}}^w\right) + \kappa\left(n_t, \Delta T\right)\left(C_{\mathcal{F}}\Delta T + C_{\mathcal{G}}\Delta T + T_{\mathcal{C}}^w\right)\right)^{-1} \\ &= n_t C_{\mathcal{F}} \frac{\left(\kappa\left(n_t, \Delta T\right) + n_t\right) T_{\mathcal{C}}^w - \left(C_{\mathcal{F}} + C_{\mathcal{G}}\right)\kappa'\left(n_t, \Delta T\right)\Delta T^2}{\left(n_t C_{\mathcal{G}}\Delta T + \left(\kappa\left(n_t, \Delta T\right) + n_t\right) T_{\mathcal{C}}^w + \left(C_{\mathcal{F}} + C_{\mathcal{G}}\right)\kappa\left(n_t, \Delta T\right)\Delta T\right)^2}. \end{aligned} \tag{7.36}$$

The optimization problem is not solvable as we do not know $\kappa\left(n_t, \Delta T\right)$, nor do we know it's derivative $\kappa'\left(n, \Delta T\right)$. We can, however, still gain insight from the above if we assume that $\lim_{\Delta T \to 0} \kappa\left(n_t, \Delta T\right) = 1$ and that $\lim_{\Delta T \to \infty} \kappa\left(n_t, \Delta T\right) = n_t$. In addition to these assumptions, we know that $\kappa'\left(n_t, \Delta T\right) \simeq 0$ for all but a select few ΔT when $\kappa\left(n_t, \Delta T\right)$ changes abruptly as the convergence criteria is accepted. From the derivative we may deduce that for some fixed $\kappa^* \in [1, n_t]$ with $\kappa'\left(n_t, \Delta T\right) = 0$, when $\Delta T \to 0$ then $\frac{\partial}{\partial(\Delta T)}\psi$ increases whilst from (7.32) we have that $\psi \to 0$. Conversely, as $\Delta T \to \infty$ then $\frac{\partial}{\partial(\Delta T)}\psi \to 0$ asymptotically. So the maximum speedup ψ must exist in the limit $\Delta T \to \infty$. Taking $\kappa'\left(n_t, \Delta T\right) = 0$ it follows that $\left(C_{\mathcal{F}} + C_{\mathcal{G}}\right)\kappa'\left(n_t, \Delta T\right)\Delta T^2 = 0$. The denominator does not change sign, so for a fixed κ^*, there are no inflection points $\Delta T \in \mathbb{R}^+$, i.e., we can not hope to find any point where the speedup decreases particularly fast as we decrease ΔT, even for some fixed κ^*. Because of these limitations, we take another approach at finding some approximate solution $\Delta \hat{T}_{opt} \approx \Delta T_{opt}$ to (7.35). While we might not be able to quantify the gain in speed-up from $\kappa\left(n_t, \Delta T\right)$ becoming smaller as we let $\Delta T \to 0$, we can quantify the associated cost of doing so in terms of added wallclock-time spent communicating $\mathbf{U}_k^n$ across node-groups. Let $\epsilon_c \in (0., 1)$ be a parameter that denotes the fraction of decrease in speedup, due to communication, that we are willing to accept. The optimization problem is then recast as

$$\Delta \hat{T}_{opt} = \min_{\Delta T \in \mathbb{R}^+} \ \Delta T \text{ s.t. } \psi\left(\Delta T\right) \geq \epsilon_c \psi_{max} \tag{7.37}$$

with some prior guess $\kappa^* \approx \kappa\left(n_t, \Delta T\right)$. The problem is now to find the smallest ΔT for which the reduced speed-up, due to communication, is less than ϵ_c the fraction of the maximal speedup ψ_{max}. The above problem is much simpler to solve. Writing out the inequality (7.37) using (7.32) one arrives at the criteria

$$\frac{n_t C_{\mathcal{F}} \Delta T}{n_t\left(C_{\mathcal{G}}\Delta T + T_{\mathcal{C}}^w\right) + \kappa^*\left(C_{\mathcal{F}}\Delta T + C_{\mathcal{G}}\Delta T + T_{\mathcal{C}}^w\right)} \geq \epsilon_c \psi_{max} \tag{7.38}$$

with

$$\psi_{max} = \lim_{\Delta T \to \infty} \frac{n_t C_{\mathcal{F}} \Delta T}{n_t \left(C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \right) + \kappa^* \left(C_{\mathcal{F}} \Delta T + C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \right)} \tag{7.39}$$

to be satisfied. We can derive an explicit expression for $\Delta \hat{T}_{opt}$ by evaluating the r.h.s limit and manipulating the inequality to find

$$\Delta \hat{T}_{opt} = \frac{\epsilon_c}{1 - \epsilon_c} \frac{\left(\kappa^* + n_t \right) T_{\mathcal{C}}^w}{\left(\kappa^* \left(C_{\mathcal{G}} + C_{\mathcal{F}} \right) + n_t C_{\mathcal{G}} \right)} \tag{7.40}$$

as an approximate solution to (7.37). In what sense does the problem (7.37) with solution (7.40) relate to the original optimization problem (7.35)? Equation (7.40) is not a solution to (7.35), rather it is intended as a rough estimate of $\Delta \hat{T}_{opt}$, that could possibly also be used over several iterations of running the algorithm to improve the estimate. The above analysis, however interesting, is somewhat artificial. Solving (7.35), we seek to optimize the parallel speed-up over a domain $T_{end} = n_t \Delta T$ that varies in size with ΔT. For parallel-in-time speedup when solving (7.1) on some (long) fixed time-domains T_{end}, we are interested in the multi-cycle Parareal introduced in the previous section. In this case $T_{end} = n_c \delta T = n_c n_t \Delta T$. We consider the number of simultaneously active time-subdomains n_t to be a fixed parameter and wish to find the optimal time-subdomain length ΔT to use. In what follows we build on the insight gained.

7.4.2 Multi Cycle Analysis

We seek an approach to estimate ΔT_{opt} for Parareal with multiple cycles, as introduced in Section 7.4.1. For the stop-restart Parareal, Figure 7.11, the parallel speed-up may be expressed as

$$\begin{aligned} \psi \left(\Delta T \right) &= \frac{n_c n_t T_{\mathcal{F}}}{n_c n_t T_{\mathcal{G}} + \sum_{i=1}^{n_c} \kappa_i \left(n_t, \Delta T \right) \left(T_{\mathcal{F}} + T_{\mathcal{G}} \right)} \\ &= \frac{n_c n_t C_{\mathcal{F}} \Delta T}{n_c n_t \left(C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \right) + \sum_{i=1}^{n_c} \kappa_i \left(n_t, \Delta T \right) \left(C_{\mathcal{F}} \Delta T + C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \right)} \end{aligned} \tag{7.41}$$

In the numerator we have the time to solve the problem sequentially applying the operator $\mathcal{F}_{\Delta T}$ $n_c n_t$ times on the initial condition. The denominator measures the time it takes when using multiple consecutive non-overlapping cycles of Parareal. Note that the above is technically an upper bound estimate as we only include the computation of $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ as well as the time to communicate a solution state $T_{\mathcal{C}}^w$, but not the time to compute the correction (5.10) and other overhead. As in the previous section, $\kappa_i \left(n_t, \Delta T \right)$ is unknown, and it is therefore not possible to solve the optimization problem (7.35) to find ΔT_{opt} a priori. As before we take the approach of solving (7.37) instead. We first let $\langle \kappa^* \rangle = \frac{1}{n_c} \sum_{i=1}^{n_c} \kappa_i$ and assume $\langle \kappa^* \rangle \approx \kappa \left(\frac{T_{end}}{n_t}, n_t \right)$ to recover

$$\frac{n_c n_t C_{\mathcal{F}} \Delta T}{n_c n_t \left(C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w\right) + n_c \left\langle \kappa^* \right\rangle \left(C_{\mathcal{F}} \Delta T + C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w\right)} \geq \epsilon_c \lim_{\Delta T \to \frac{T_{end}}{n_t}} \psi\left(\Delta T\right) \tag{7.42}$$

where

$$\epsilon_c \lim_{\Delta T \to \frac{T_{end}}{n_t}} \psi\left(\Delta T\right) = \frac{\epsilon_c n_t C_{\mathcal{F}} T_{end}}{n_t \left(C_{\mathcal{G}} T_{end} + n_t T_{\mathcal{C}}^w\right) + \left\langle \kappa^* \right\rangle \left(C_{\mathcal{F}} T_{end} + C_{\mathcal{G}} T + n_t T_{\mathcal{C}}^w\right)} \tag{7.43}$$

With $n_c = \frac{T_{end}}{n_t \Delta T}$, the solution to (7.37) becomes

$$\Delta \hat{T}_{opt}^{\beta=0} = \frac{\epsilon_c \left(n_t + \left\langle \kappa^* \right\rangle\right) T_{\mathcal{C}}^w T_{end}}{\left(n_t + \left\langle \kappa^* \right\rangle\right) n_t T_{\mathcal{C}}^w + \left(1 - \epsilon_c\right) \left(n_t C_{\mathcal{G}} + \left(C_{\mathcal{F}} + C_{\mathcal{G}}\right) \left\langle \kappa^* \right\rangle\right) T_{end}} \tag{7.44}$$

In the case of the Adaptive Parareal, Figure 7.12, consecutive cycles are allowed to overlap in execution time across nodes. For the case of the “impatient” algorithm, $\beta = 1$, this will ideally hide the initialization cost of the zero’th iteration for all but the first cycle. In this case we may approximate the speed-up as

$$\psi\left(\Delta T\right) = \frac{n_c n_t C_{\mathcal{F}} \Delta T}{n_t \left(C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w\right) + \sum_{i=1}^{n_c} \kappa_i \left(n_t, \Delta T\right) \left(C_{\mathcal{F}} \Delta T + C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w\right)}. \tag{7.45}$$

Note that (7.45) is no longer equivalent to (7.32) for the single cycle case. We take another look at the derivative for ideas on how to solve (7.35).

$$\frac{\partial}{\partial\left(\Delta T\right)} \psi\left(\Delta T\right) = -T_{end} C_{\mathcal{F}} \frac{n_t C_{\mathcal{G}} + \frac{T}{n_t} \left(\left\langle \kappa^* \right\rangle' \left(C_{\mathcal{F}} + C_{\mathcal{G}} + \frac{T_{\mathcal{C}}^w}{\Delta T}\right) - \left\langle \kappa^* \right\rangle \frac{T_{\mathcal{C}}^w}{\Delta T^2}\right)}{\left(n_t \left(C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w\right) + \frac{T}{n_t} \left\langle \kappa.^* \right\rangle \left(C_{\mathcal{F}} + C_{\mathcal{G}} + \frac{T_{\mathcal{C}}^w}{\Delta T}\right)\right)^2}. \tag{7.46}$$

As before we assume $\left\langle \kappa^* \right\rangle' = 0$, it is clear that if $\sqrt{\left\langle \kappa^* \right\rangle T_{end} T_{\mathcal{C}}^w n_t^{-2} C_{\mathcal{G}}^{-1}} = \Delta T \leq \frac{T_{end}}{n_t}$ then the derivative $\psi\left(\Delta T\right)$ changes sign somewhere in $\left(0, \frac{T_{end}}{n_t}\right]$, e.g., the largest value of $\psi\left(\Delta T\right)$ is not necessarily in the limit $\Delta T \to \frac{T_{end}}{n_t}$. With $\frac{\partial}{\partial(\Delta T)} \psi\left(\Delta T\right) = 0$, only a single maxima exists for ΔT in $\mathbb{R}^+$

$$\Delta \hat{T}_{max} = \sqrt{\frac{\left\langle \kappa^* \right\rangle T_{end} T_{\mathcal{C}}^w}{n_t^2 C_{\mathcal{G}}}}. \tag{7.47}$$

Inserting (7.47) into (7.45) and using that $n_c = \frac{T_{end}}{n_t \Delta T}$, the associated speedup is

$$\begin{aligned} \psi_{max} &= \frac{n_c n_t C_{\mathcal{F}} \Delta T_{max}}{n_t \left(C_{\mathcal{G}} \Delta T_{max} + T_{\mathcal{C}}^w \right) + n_c \left\langle \kappa^* \right\rangle \left(C_{\mathcal{F}} \Delta T_{max} + C_{\mathcal{G}} \Delta T_{max} + T_{\mathcal{C}}^w \right)} \\ &= \frac{n_t C_{\mathcal{F}} T_{end}}{T_{end} \left\langle \kappa^* \right\rangle \left(C_{\mathcal{F}} + C_{\mathcal{G}} \right) + n_t^2 T_{\mathcal{C}}^w + 2 \sqrt{n_t^2 T_{\mathcal{C}}^w \left\langle \kappa^* \right\rangle C_{\mathcal{G}} T_{end}}}. \end{aligned} \tag{7.48}$$

With ψ_{max} as above, the constraint (7.37) to be satisfied may be written as

$$\frac{n_c n_t C_{\mathcal{F}} \Delta T}{n_t \left(C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \right) + n_c \left\langle \kappa^* \right\rangle \left(C_{\mathcal{F}} \Delta T + C_{\mathcal{G}} \Delta T + T_{\mathcal{C}}^w \right)} \geq \epsilon_c \psi_{max}. \tag{7.49}$$

Inserting (7.48) into (7.49) and manipulating the inequality we recover

$$\frac{\epsilon_c n_t^2 C_{\mathcal{G}} \Delta T^2 + \epsilon_c \left\langle \kappa^* \right\rangle T_{\mathcal{C}}^w T_{end}}{\left(2 \sqrt{n_t^2 \left\langle \kappa^* \right\rangle T_{\mathcal{C}}^w C_{\mathcal{G}} T_{end}} + (1 - \epsilon_c) \left(n_t^2 T_{\mathcal{C}}^w + \left\langle \kappa^* \right\rangle \left(C_{\mathcal{F}} + C_{\mathcal{G}} \right) T_{end} \right) \right) \Delta T} \leq 1 \tag{7.50}$$

from which we recover the smallest ΔT, satisfying the quadratic inequality, is

$$\Delta \hat{T}_{opt}^{\beta=1} = \Delta \hat{T}_{\epsilon} - \sqrt{\left(\Delta T_{\epsilon} \right)^2 - \Delta \hat{T}_{max}^2} \tag{7.51}$$

where

$$\Delta \hat{T}_{\epsilon} = \frac{1}{\epsilon_c} \Delta \hat{T}_{max} + \frac{1 - \epsilon_c}{2 \epsilon_c} \left(\frac{T_{\mathcal{C}}^w}{C_{\mathcal{G}}} + \frac{1}{T_{\mathcal{C}}^w} \Delta \hat{T}_{max}^2 \left(C_{\mathcal{F}} + C_{\mathcal{G}} \right) \right) \tag{7.52}$$

and $\Delta \hat{T}_{max}$ is defined in (7.47). This yields a rough estimate of $\Delta \hat{T}_{opt}^{\beta=1}$ that ensures that the choice of ΔT is not too long such as to waste communication bandwidth, while ensuing that moving data does not become a new significant limitation to parallel acceleration. In deriving $\Delta \hat{T}_{opt}^{\beta}$ for the adaptive scheduler with $\beta \to 1$, i.e. impatient, it was assumed that the initialization procedure of the zero'th iteration was perfectly hidden by the adaptive scheduler for all but the first cycle as depicted in Figure 7.12b. This is unlikely to be the case for all cycles, and we therefore expect that the optimal ΔT will lie somewhere in the interval $\left[\Delta \hat{T}_{opt}^{\beta=0}, \Delta \hat{T}_{opt}^{\beta=1} \right]$ spanned by (7.44) and (7.51).

In section 7.5, parallel scaling is measured when using each scheduler with a time-subdomain length as estimated from (7.44) and (7.51). To somehow test the effectiveness of the estimates derived, one would need to measure speed-up for all possible different time-subdomain lengths for a given problem to compare if the peak speedup occurs for the $\Delta \hat{T}$ predicted.

Doing so for the high-order 2D shallow water equation solver introduced as the test-case in section 7.2.3 is computationally intractable. Instead we perform a test on a smaller numerical experiment by computing an approximation to the scalar complex initial value problem (7.30) also used in the previous section to investigate how the overlapping adaptive scheduler change the convergence pattern of the algorithm. Again we let $dt = 10^{-5}$ and $dT = 10^{-3}$ in $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ respectively.

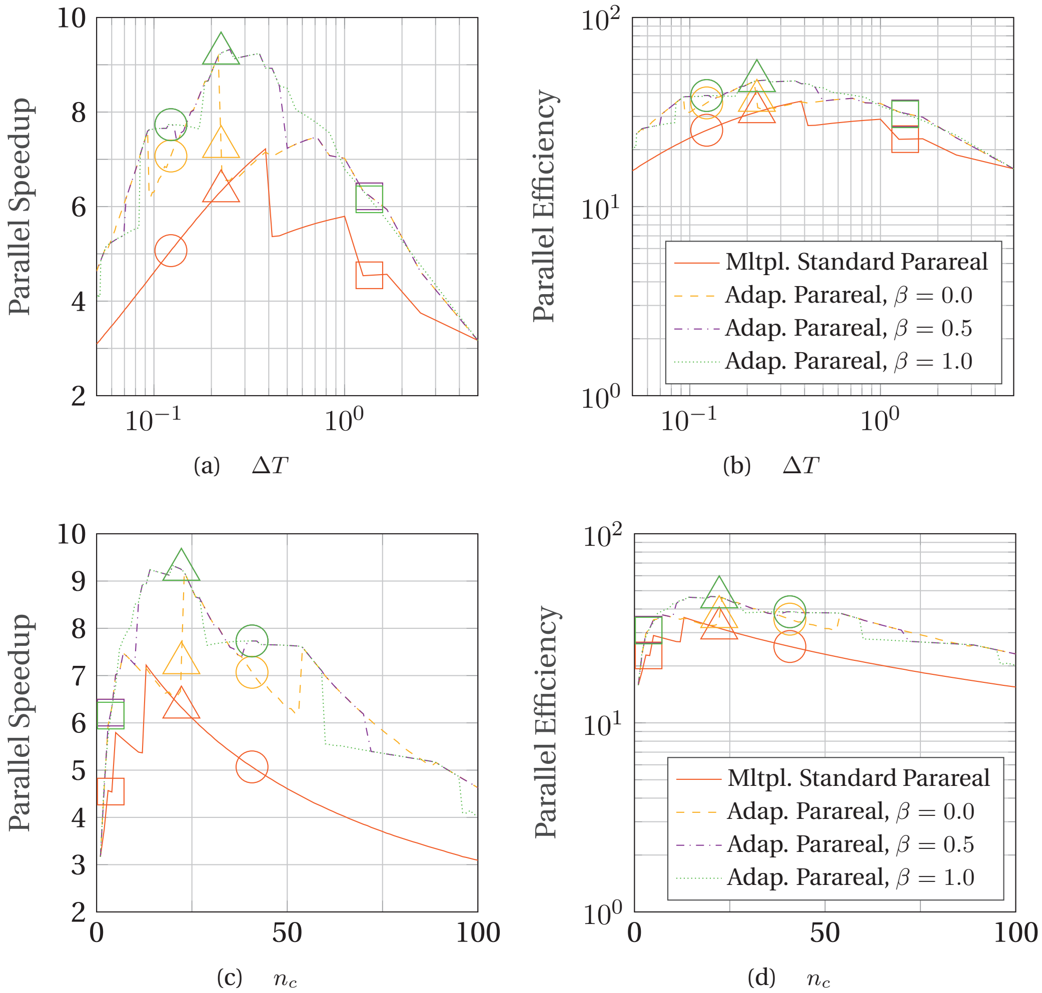


Figure 7.14 – Parallel speedup and efficiency measured when solving (7.30) as a function of time-subdomain length ΔT (a)-(b), and number of cycles n_c (c)-(d). Using $n_t = 20$ processors parallel-in-time and letting $C_{\mathcal{F}} = 10000\text{ms}$, $C_{\mathcal{G}} = 100\text{ms}$, $T_c^w = 100\text{ms}$. The square, circle and triangle markers indicate the estimates for the optimal time-subdomain length using (7.44), (7.51) and the average of the two, respectively.

For the test we perform the integration until $T_{end} = 100$ with $n_t = 20$ simultaneously active time-subdomains on 20 processors. The test is performed using from $n_c = 1$ to $n_c = 100$ cycles, corresponding to time-subdomain lengths from $\Delta T = 10^{-3}$ to $\Delta T = 10^{-1}$. A time-subdomain is accepted as converged if the norm on the difference between two consecutive iterations is smaller than some tolerance and if all previous time-subdomains have converged. Here we use the tolerance $\epsilon = 0.01$. Since the application of both $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ only require 6 floating point operations, the wall-time used by a processors is set manually to $C_{\mathcal{F}} = 10000$ms, $C_{\mathcal{G}} = 100$ms, and communication of a time-subdomain interval to $T_c^w = 100$ms.

The numerical experiments are presented in Figure 7.14. The estimates for the time-subdomain length that will result in the highest speedup are indicated by the square, triangle and circle markers. The estimates were computed using a priori guess $\langle \kappa^* \rangle = 2$ and limit $\epsilon_c = 95\%$. We note that the estimates are not sharp, but nevertheless the optimal time-subdomain length is smaller than the upper estimate and larger than the lower estimate. Using the average of the two estimates appear to give the best prediction on the optimal time-subdomain lengths for large speedup. In particular we note that the ΔT, resulting in the highest speedup for the adaptive scheduler was measured to be very close to the arithmetic average of n_c from the estimates (7.44) and (7.51). This indicates that the estimates may indeed be useful for approximating the optimal choice of time-subdomain length a priori.

Appendix C contains 6 figures of tests like those presented in Figure 7.14. Here it is investigated how well the estimates from equation (7.44) and (7.51) approximate the optimal time-subdomain length is tested for communication cost $T_c^w = 1ms, 10ms, 100ms$, and three different coarse operator time-step lengths. The cost of performing the coarse operator $C_{\mathcal{G}}^w$ is made proportional to it's time-step length. Results presented both as a function of number of cycles N_c, and as a function of time-subdomain length. We observe good agreement between the predicted values and the actual optimal value. The estimates are not sharp, but using them it appears that one may safely avoid making the time-subdomains much too long or much too short.

7.5 Numerical Experiments: Parallel Scaling

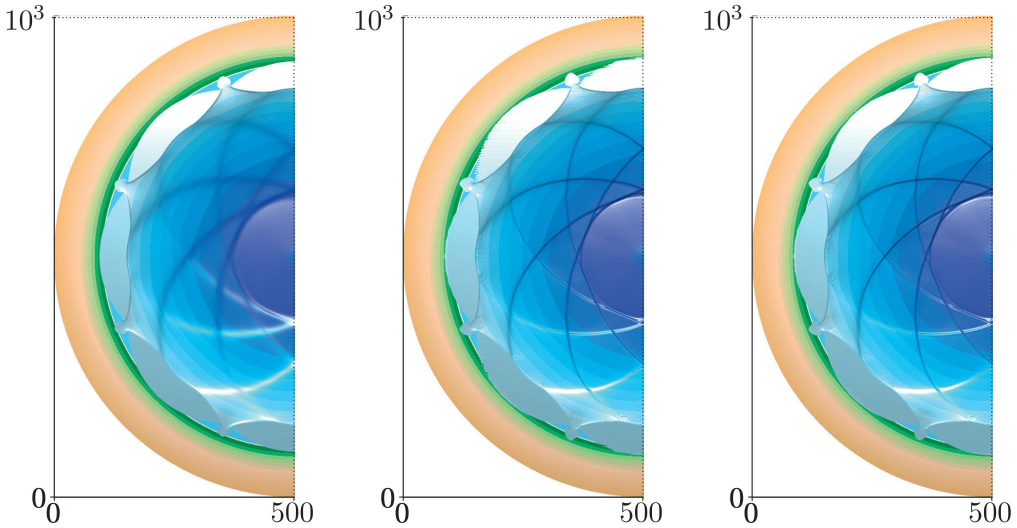
Numerical experiments using time-parallel integration on the test case, introduced in section 7.2.3, are presented in the following. For all numerical experiments, the EPFL Bellatrix general purpose cluster consisting of 424 compute nodes, each with two 8-core Intel Xeon Sandy Bridge processors and infiniband QDR 2:1 network has been used. The adaptive work scheduler, combined with the approach of estimating the time-subdomain length that optimally balances communication cost and convergence speed, is collectively denoted as CAAP. In all cases, an approximation to (7.1) is being computed on an interval of length $T_{end} = 60$min. A tolerance of $\epsilon = 10^{-4}$ on the norm of the difference between consecutive iterations is used as a convergence criteria. To evaluate if the solution, found iteratively trough Parareal or CAAP, is as accurate as the sequential WENO SSP-ERK solution, we define two errors measurements. The relative error with respect to the sequential WENO SSP-ERK solution

$$\tilde{\varepsilon}\left(U_{n_t n_c}^{k_{conv}}\right) = \frac{\sqrt{\int_\Omega \left|U_{n_t n_c}^{k_{conv}} - \mathcal{F}_{\Delta T}^{n_t n_c} u\left(\cdot, T_0\right)\right|^2 d\Omega}}{\int_\Omega \mathcal{F}_{\Delta T}^{n_t n_c} u\left(\cdot, T_0\right) d\Omega} \tag{7.53}$$

and the relative error with respect to the true solution

$$\hat{\varepsilon}\left(U_{n_t n_c}^{k_{conv}}\right) = \frac{\sqrt{\int_\Omega \left|U_{n_t n_c}^{k_{conv}} - u\left(\cdot, T_{end}\right)\right|^2 d\Omega}}{\int_\Omega u\left(\cdot, T_{end}\right) d\Omega} \tag{7.54}$$

No analytical solution is known to exists for our test-case. We therefore approximate the "true" solution using a very fine mesh with 14400x14400 cells with adaptive time-steps, computed as outlined in section 7.2.1.



(a) (b) (c)

Figure 7.15 – Water surface at $T = 60$min starting with the initial condition described in Section 7.2.3 at $T = 0$min as computed when using (a) Roe's method (b) WENO SSP-ERK (c) CAAP with tolerance $\epsilon = 10^{-4}$ on the norm of the difference between two consecutive iterations. The coarse operator captures wave speeds very well but introduces dissipative errors. From a bird-eye perspective it appears that CAAP arrives at the same solution as the parallel WENO SSP-ERK with sequential time-stepping. In Figure 7.16 a zoomed in view of the profile around a shock is presented. Units in kilometers.

The relative error $\hat{\varepsilon}$ between the WENO SSP-ERK approximation and the true solution at $T = 60min$ is $3.7 \cdot 10^{-3}$. For Roe's method, used here as a preconditioner in Parareal and CAAP, the relative accuracy $\hat{\varepsilon}$ is $1.2 \cdot 10^{-2}$.

Ideally, the error of the Parareal solution and the CAAP solution with respect to the true solution, should be very close to that of the sequential WENO SSP-ERK solution. In the numerical experiments presented in Table 7.1, we measure $\hat{\varepsilon}$ to be between $3.7 \cdot 10^{-3}$ and $3.9 \cdot 10^{-3}$ for CAAP, and between $4.0 \cdot 10^{-3}$ and $4.1 \cdot 10^{-3}$ for the standard Parareal algorithm. In Figure 7.15 the water height at $T = 60min$ for the sequential time-stepping WENO SSP-ERK scheme is depicted along with the Parareal solution. Upon further investigation we find that the maximum error of the Parareal solution is always in the vicinity of shocks. See Figure 7.16 for a cross-section view of the solution in Figure 7.15. Here the Parareal solution is given for different tolerances.

To estimate the number of Parareal cycles to use in CAAP, i.e. the time-subdomain length ΔT, we need to measure the constants $C_{\mathcal{F}}$, $C_{\mathcal{G}}$ and $T_{\mathcal{F}}^{w}$ on the EPFL Bellatrix cluster on which we will run our numerical experiments. We run the code over the full interval of 60 minutes to estimate the two ratios $C_{\mathcal{F}}$ and $C_{\mathcal{G}}$. Doing so we find that we need roughly 4000ms to compute 1 minute of simulation time when using 5 nodes in space. For the coarse operator, we need roughly 300ms.

$$C_{\mathcal{F}} = 4018.3\text{ms/min}, \qquad C_{\mathcal{G}} = 309.4\text{ms/min}, \qquad T_{\mathcal{F}}^{w} = 75\text{ms} \tag{7.55}$$

Measuring the time used for transferring a complete solution state across a time-subdomain interface from one group of nodes to another, denoted $T_{\mathcal{F}}^{w}$, proved difficult as the number fluctuates substantially, depending on the load of the cluster and of the location of the nodes. Doing multiple runs on different node locations, we arrived at 75ms on average when using 5 nodes in each group. In comparison it takes around 15ms to exchange the halo on the 1600x1600 cell test-case mesh. We let $\epsilon_c = 0.95$ and $\langle k^* \rangle = 2$, and use these measured quantities with (7.44) and (7.51) to estimate the optimal number of cycles n_c to split the time domain into.

The values are listed in Table 7.1, the high estimates computed using (7.51) and the low using (7.44), the estimates are rounded. The parallel speed-up of the average number of cycles between the two estimates is also tested. As can be seen from the numbers in the table, it appears to generally give the highest speed-up to use the average of the two estimates.

In Figure 7.17, the equivalent scaling measurements of the space+time parallel code is presented. The maximum attainable speed-up for the conventional parallel-in-space WENO SSP-ERK implementation is 49 using 5 nodes (80 cores). Using 6 nodes and above, no further speedup is possible. Adding CAAP with $\beta = 0.5$ and $n_t = 24$, we measure a speedup of 229 using 1920 cores. The factor 4.5 reduction in time to solution of the test-case is substantial and may be critical in cases where a simulation is expected to take days. It is also worth noticing that for CAAP with $\beta = 0.5$ and $n_t = 8$, we measure a parallel-in-time efficiency of almost 35%. This is the highest parallel efficiency that has been demonstrated for parallel-in-time integration of a purely hyperbolic nonlinear PDE system using domain decomposition in time. We conjecture that for less challenging problems, e.g. diffusion dominated, still higher efficiency may be possible using the proposed method.

Appendix D contains figures with drawings of the work schedulers when solving the shallow water wave equation. Unlike the drawings of the work schedulers in Figures 7.10, 7.11, and 7.12 that are purely conceptual and included to ease the introduction of the algorithms, all figures in appendix D are created from time-stamps written to a file during the actual solution procedure on the cluster.

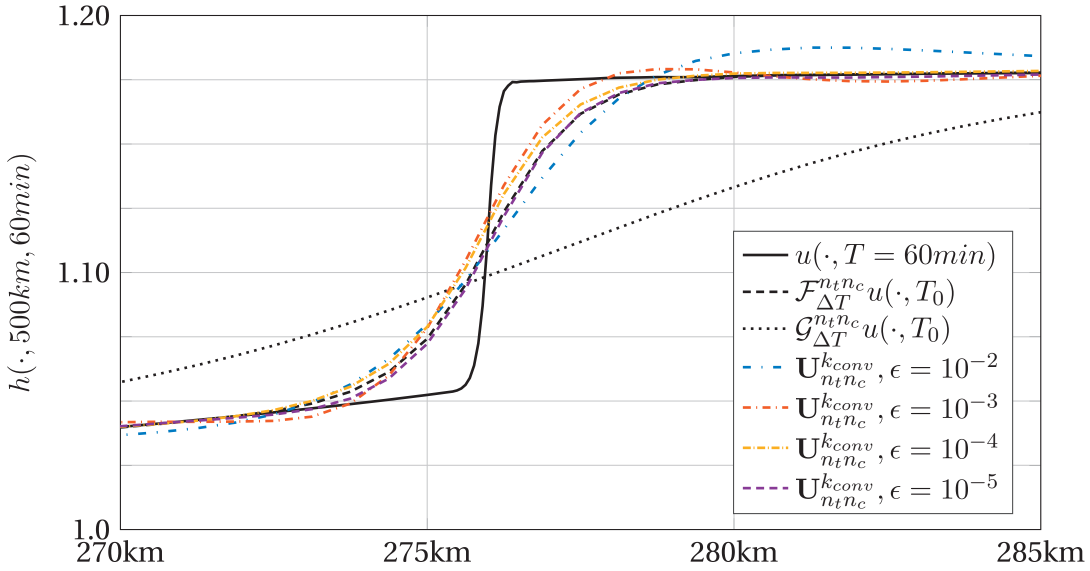
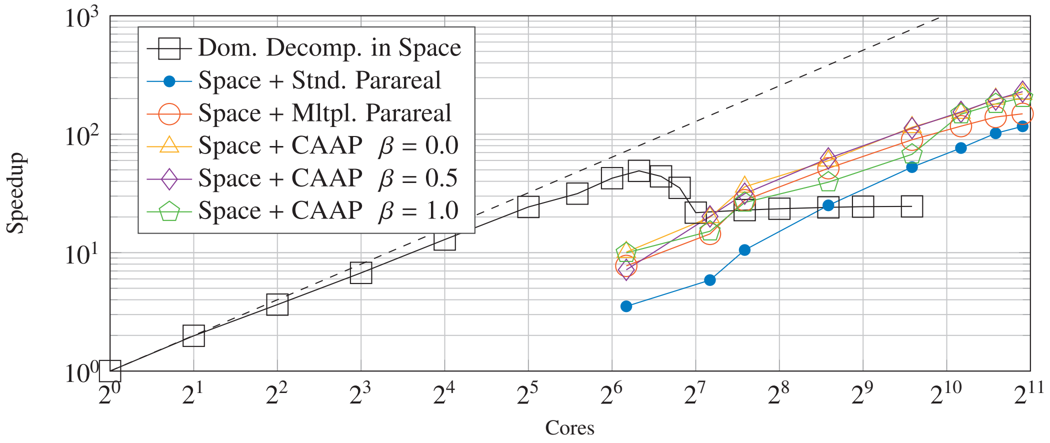


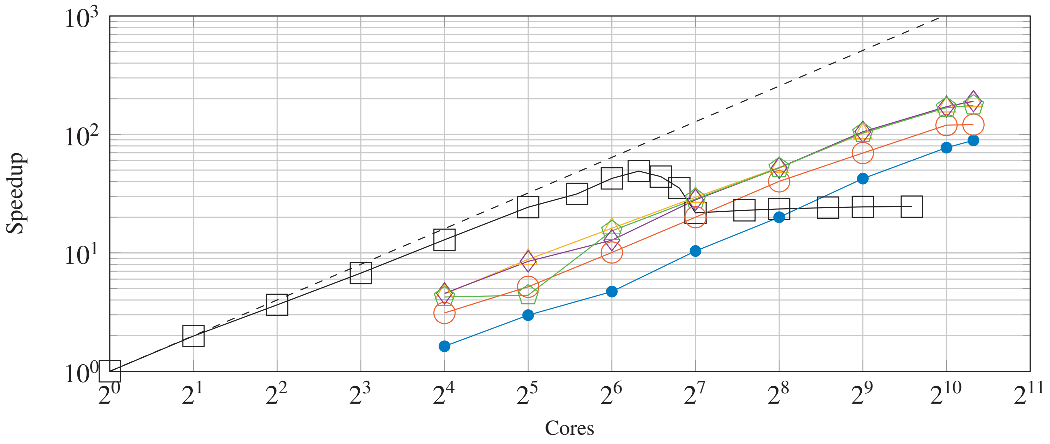
Figure 7.16 – Cross section of the water height at $y = 500km$ where two shocks meet, see figure 7.15. The cell width is $dx = 0.625$km, so the above profile is resolved with 24 cells. The error of the Parareal solution is largest in magnitude around shocks. The time-to-solution using the fine and the coarse solver on a single node, sequentially in time, is respectively 912s and 79s. For CAAP, using a single node in space on each of 16 simultaneously active time-subdomains, the time-to-solution is 205s, 199s, 242s and 361s respectively. It is somewhat surprising that for the largest tolerance we do not measure the shortest time-to-solution. Looking at the log files of the simulation, it appears that this is due to less effective load balancing and that in some intervals convergence happens in just 1 iteration, i.e. second correction, whilst in other time-subdomains it takes several iterations despite the large tolerance.

Type, $n_t = 8$	n_c	ΔT $[s]$	$\tilde{\varepsilon}$	$\hat{\varepsilon}$	P. Eff. T $[\%]$	P. Eff. T+S $[\%]$	P. Speedup
Stnd. Parareal	1	450	$1.4 \cdot 10^{-3}$	$4.1 \cdot 10^{-3}$	14.9	9.1	58.4
Mltpl. Parareal	5	90	$1.4 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	18.3	11.2	71.8
CAAP, $\beta = 0.0$	5	90	$1.4 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	25.5	15.6	100.1
CAAP, $\beta = 0.5$	5	90	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	22.2	13.6	87.2
CAAP, $\beta = 1.0$	5	90	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	22.0	13.5	86.4
Mltpl. Parareal	20	22.5	$1.4 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	27.1	16.6	106.4
CAAP, $\beta = 0.0$	20	22.5	$1.4 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	27.9	17.1	109.5
CAAP, $\beta = 0.5$	20	22.5	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	34.1	20.9	133.8
CAAP, $\beta = 1.0$	20	22.5	$1.5 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	31.7	19.4	124.4
Mltpl. Parareal	40	11.25	$1.4 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	24.8	15.2	97.0
CAAP, $\beta = 0.0$	40	11.25	$1.4 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	25.8	15.8	101.2
CAAP, $\beta = 0.5$	40	11.25	$1.4 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	31.8	19.5	125.1
CAAP, $\beta = 1.0$	40	11.25	$1.5 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	29.4	18.0	115.1
Type, $n_t = 16$	n_c	ΔT $[s]$	$\tilde{\varepsilon}$	$\hat{\varepsilon}$	P. Eff. T $[\%]$	P. Eff. T+S $[\%]$	P. Speedup
Stnd. Parareal	1	225	$1.4 \cdot 10^{-3}$	$4.1 \cdot 10^{-3}$	11.1	6.8	86.8
Mltpl. Parareal	3	75	$1.5 \cdot 10^{-3}$	$4.0 \cdot 10^{-3}$	8.7	5.3	67.3
CAAP, $\beta = 0.0$	3	75	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	11.9	7.3	93.7
CAAP, $\beta = 0.5$	3	75	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	12.1	7.4	95.0
CAAP, $\beta = 1.0$	3	75	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	11.6	7.1	90.3
Mltpl. Parareal	15	15	$1.5 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	15.5	9.5	120.9
CAAP, $\beta = 0.0$	15	15	$1.6 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	23.8	14.6	187.4
CAAP, $\beta = 0.5$	15	15	$1.6 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	23.8	14.6	186.7
CAAP, $\beta = 1.0$	15	15	$1.7 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	21.2	13.0	166.9
Mltpl. Parareal	15	7.5	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	17.1	10.5	134.0
CAAP, $\beta = 0.0$	30	7.5	$1.7 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	22.4	13.7	175.8
CAAP, $\beta = 0.5$	30	7.5	$1.6 \cdot 10^{-3}$	$3.7 \cdot 10^{-3}$	23.3	14.3	182.5
CAAP, $\beta = 1.0$	30	7.5	$1.7 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	22.5	13.8	176.9
Type, $n_t = 24$	n_c	ΔT $[s]$	$\tilde{\varepsilon}$	$\hat{\varepsilon}$	P. Eff. T $[\%]$	P. Eff. T+S $[\%]$	P. Speedup
Stnd. Parareal	1	150	$1.5 \cdot 10^{-3}$	$4.0 \cdot 10^{-3}$	10.1	6.2	118.6
Mltpl. Parareal	3	50	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	7.3	4.5	85.6
CAAP, $\beta = 0.0$	3	50	$1.6 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	9.0	5.5	106.0
CAAP, $\beta = 0.5$	3	50	$1.6 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	8.5	5.2	98.8
CAAP, $\beta = 1.0$	3	50	$1.6 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	8.5	5.2	99.3
Mltpl. Parareal	15	10	$1.5 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	13.1	8.0	153.7
CAAP, $\beta = 0.0$	15	10	$1.7 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	18.6	11.4	218.0
CAAP, $\beta = 0.5$	15	10	$1.7 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	19.4	11.9	228.6
CAAP, $\beta = 1.0$	15	10	$1.7 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	17.1	10.5	201.0
Mltpl. Parareal	25	6	$1.6 \cdot 10^{-3}$	$3.7 \cdot 10^{-3}$	11.4	7.9	151.3
CAAP, $\beta = 0.0$	25	6	$1.8 \cdot 10^{-3}$	$3.7 \cdot 10^{-3}$	17.1	10.5	201.7
CAAP, $\beta = 0.5$	25	6	$1.7 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	18.6	11.4	218.4
CAAP, $\beta = 1.0$	25	6	$1.9 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	14.7	9.9	189.4

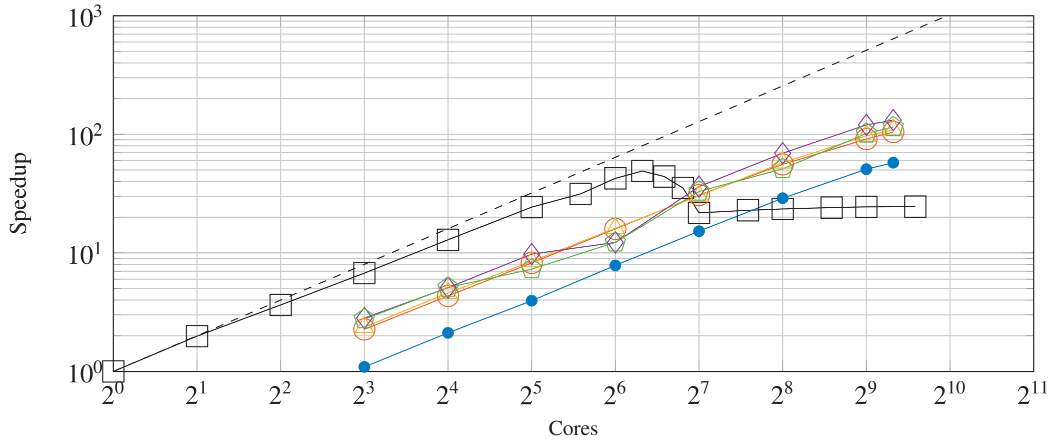
Table 7.1 – Parallel acceleration as measured using $n_t = \{8, 16, 24\}$ simultaneously active time-subdomains. Within each time-subdomain, 5 nodes (80 cores) are used parallel in space, for a total of $\{640, 1280, 1920\}$ cores, respectively. The first parallel efficiency column indicates the efficiency of the parallel-in-time integration procedure, the second column indicates the combined space-time parallel efficiency.



(a) Using $n_t = 24$ simultaneously active time-subdomains



(b) Using $n_t = 16$ simultaneously active time-subdomains



(c) Using $n_t = 8$ simultaneously active time-subdomains

Figure 7.17 – Speedup as a function of number of cores used in the space-time parallel code for solving the SHW test case introduced in Section 7.2.3 as measured on the 424-node EPFL Bellatrix cluster. $T_{end} = 60\text{min}$, and $n_c = \{20, 15, 15\}$ for $n_t = \{8, 16, 24\}$ respectively. Tolerance $\epsilon = 10^{-4}$ was used as the convergence criteria.

In the application the finite volume method with a Roe's flux used as a coarse operator, is roughly 12 times faster than the WENO SSP-ERK fine solver. This means that no matter how many simultaneously active time-subdomains we use, we will never gain more than a factor 12 reduction in time to solution with respect to a pure space parallel WENO SSP-ERK implementation. We did attempt to use a cheaper coarse operator, testing the Lax-Friedrichs method with a diffusion term for stability. This operator was roughly 40 times faster than the WENO SSP-ERK scheme, potentially allowing for much create parallel-in-time speed-up. Unfortunately, we observed that in order for the algorithm to convergence, the length of the time-subdomain had to be as short as a single time-step, otherwise the algorithm would diverge for a couple of iterations before eventually converging, leading to no, or very little, speedup. With a time-subdomain length equivalent to a single time-step, little parallel-speed-up was possible, in excess of spatial saturation, because the full spatial solution state must then be transfered from one group of nodes to another in between every timestep.

7.6 Summary

The Parareal algorithm is one among several new algorithms that seek to introduce parallelism in time. Unlike other proposed models, Parareal has the distinct advantage of being potentially highly scalable and minimally invasive. The algorithm may, to a large extent, simply be wrapped around existing simulation codes in combination with the introduction of some coarse operator. The parallel efficiency of the method, particularly for hyperbolic problems and for integration of long time domains has however been a major concern. In this chapter we have demonstrated parallel-in-time efficiency of upwards of 35% for long time integration of a purely hyperbolic problem using CAAP. This is more than double of what could be achieved using the standard Parareal approach with the scheduler presented in [9]. With CAAP, we demonstrated a speedup of 228 in our space-time parallel tests, compared to a maximum speedup of 49 for the original code. The factor of 4.5 reduction in time to solution of the test-case is substantial and may be critical in cases where a simulation is expected to take days or if real-time response is needed.

For the coarse operator $\mathcal{G}_{\Delta T}$, we used a lower order discretization that operates on the same mesh as the original WENO SSP-ERK solver. The choice was motived by experience gathered by testing multiple different coarse operators on a simpler 1D problem. In our experience, for Parareal to converge on a hyperbolic problem, one must design the coarse operator in such a way that it only introduces dissipative errors with respect to the fine operator. In addition, we observed during these preliminary tests that designing a coarse operator in such a way that it is as accurate as possible under the constraint that $C_{\mathcal{F}}/C_{\mathcal{G}} > n_t$ is satisfied seems to lead to the highest overall speedup.

For all our numerical experiments, we have used a tolerence on the norm of the difference between two consecutive iterations as a convergence criteria. This is, however, not necessarily a good predictor of the actual error. We therefore presented error measurements along with all speedup measurements to document that the error of the Parareal solution is indeed as small as the error of the sequential solution. Recently, techniques for posteriori error analysis on Parareal have been developed[60], and the approach may provide a way to certify a priori that the solution found is sufficiently accurate.

While we consider our findings to offer substantial progress in the understanding of how to move past the strong scaling saturation limit of classical spatial domain-decomposition methods, there are potential improvements to be made. Using CAAP, we have observed that small load imbalances arise due to the nature of the test-case. In the nonlinear shallow water wave equation solver in which we implemented time-parallel integration, time-steps are adaptive. In between the integration of each timestep, the longest possible next timestep is computed and therefore some time-subdomains end up using a different number of time-steps than others. This in turn creates small load imbalances which may only be partially hidden by the adaptive scheduler. We speculate that a better approach may be to use the proposed optimal time-subdomain estimates derived in Section 7.4 to compute an optimal wallclock-time and from this set a fixed number of time-steps to use in a time-subdomain during the zero'th iteration of the preconditioner, i.e., the decomposition in time is not fixed, but computed dynamically and set during the zeroth iteration for each time-subdomain as the parallel-in-time integration procedure progresses. In this way one may achieve significantly better load-balancing for problems with an adaptive timestep integrator, and in doing so the proposed scheduler would be adaptive, not only in balancing utilization and convergence, but also in setting the length of each time-subdomain dynamically during the integration procedure.

Finally, we note that Parareal and CAAP as methods for domain-decomposition in time are different from classical (spatial) domain decomposition in the sense that several copies of the solution states at time-subdomain interfaces must be stored. This extra use of memory is not of primary concern since Parareal is of interest in the strong scaling limit. In [231] it was, however, demonstrated how these extra states may be used to make the algorithm in [9] tolerant to node failures, even when the underlying operators $\mathcal{F}_{\cdot\mathcal{T}}$ and $\mathcal{G}_{\cdot\mathcal{T}}$ themselves are not. Since the adaptive scheduler proposed in Section 7.3 is based on overlapping cycles of the same scheduler, we expect that it too may be made tolerant towards node failures in a similar manner.

8 Time-Parallel Integration for Convection Dominated Problems

In chapter 6 it was discussed how, and why, Parareal tends to perform poorly on convection dominated PDE problems. The horizontal movement of dispersive error components in the coarse operator $\mathcal{G}_{\Delta T}$, that differ from those in $\mathcal{F}_{\Delta T}$, are wrongly interpreted as a need for vertical correction which causes errors to be introduced and amplified in subsequent iterations. This behavior limits the applicability of Parareal to only certain hyperbolic and convection dominated problems where it is possible to create a cheaper coarse operator $\mathcal{G}_{\Delta T}$ that essentially resolve all wave behavior of $\mathcal{F}_{\Delta T}$ accurately. In this chapter we first present in Section 8.1 two approaches at modifying the Parareal algorithm to facilitate correction of phase errors. This is followed in Section 8.2 with the presentation of a new way of deriving parallel-in-time integration schemes. To illustrate the idea, a scheme is derived and tested.

8.1 Modifying Parareal

In section 8.1, we present two experiments of modifying the Parareal algorithm to introduce correction on phase errors. In section 8.1.1, numerical experiments of doing Parareal-style phase correction in Fourier space is presented, and in section 8.1.2, a more generally applicable local correction procedure is developed and tested.

8.1.1 Fourier Space Phase Correction

As a first experiment, we investigate the impact on convergence of performing the Parareal correction in Fourier space. At each iteration, the Parareal correction (5.10), using $\mathcal{G}_{\Delta T}^{T_n}U_n^k$, $\mathcal{F}_{\Delta T}^{T_n}U_n^k$ and $\mathcal{G}_{\Delta T}^{T_n}U_n^{k+1}$ to compute U_{n+1}^{k+1}, is performed independently twice, once to correct all wave numbers, once to correct all amplitude coefficients. We test the alternative correction procedure on the 1D constant coefficient linear advection equation (6.1) with periodic boundary conditions. Both $\mathcal{F}_{\Delta T}^{T_n}$ and $\mathcal{G}_{\Delta T}^{T_n}$ are constructed using a fourth ordered compact finite difference stencil[216], $\mathcal{G}_{\Delta T}^{T_n}$ operates on a space-time mesh coarsened by a factor 5. The numerical experiments are presented in figure 8.1, using the discontinuous initial condition from chapter 6.

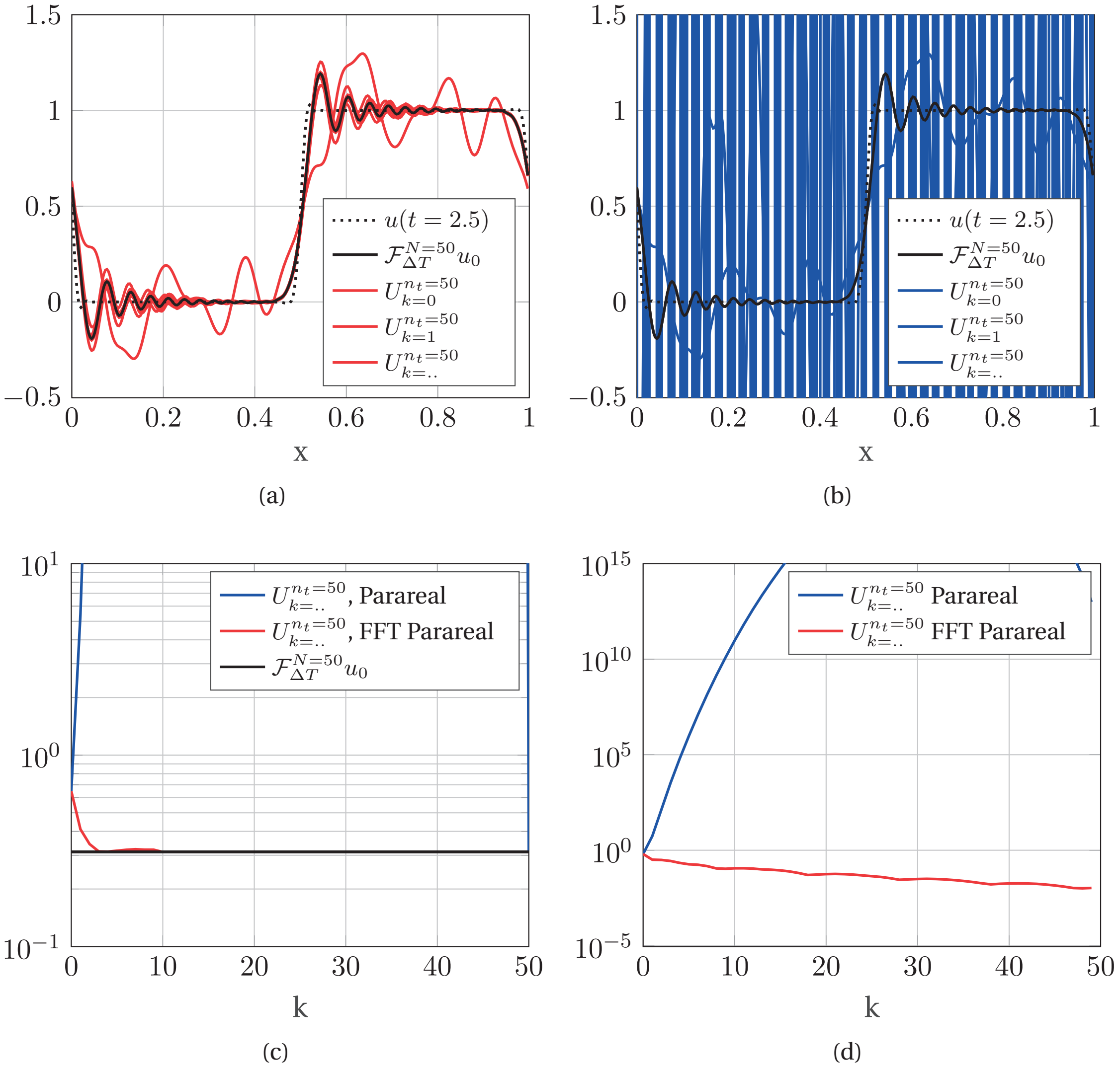


Figure 8.1 – The application of Parareal for parallel-in-time integration, on $n_t = 50$ time-subdomains, of the advection-diffusion equation with $\kappa = 10^{-5}$ and $a = 1$ to $T = 2.5$ using the initial condition in figure 6.1a. The dispersive operators $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ are created using 301 and 61 points in space respectively, with a 4th order compact finite difference scheme. Figures (a) and (b) shows the sequential solution $\mathcal{F}_{\Delta T}^{N=50}u_0$, the true solution $u(t=2.5)$, and consecutive iterations $U_k^{N=50}$. In figure (a), correction is performed in Fourier space as outlined in section 8.1.1, in figure (b) regular Parareal corrections are used. (c) shows the error at $t = 2.5$ as measured with respect to the true solution. (d) shows the error as measured with respect to the fine solution. All errors are measured in the infinity norm on the entire space-time domain.

The test presented in Figure 8.1 is not an easy test in the context of parallel-in-time integration. The diffusion constant in (6.1) is very small, the shape to advect is discontinuous, and crucially, the discretization is dispersive. From the results it is however clear that using Parareal to correct wave modes, individually on both phase and amplitude components, stabilize the algorithm to the point that monotone convergence is observed. Even though the approach of correcting twice in Fourier space appears to very work well, it is of limited practical relevance since there is no obvious extension of the method for anything else than linear constant coefficient hyperbolic problems with periodic boundary conditions. It however serves as another demonstration of how the traditional Parareal algorithm is unable to cope with phase error differences between $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$, and it indicates that the community should seek to create new methods without this inherent limitation.

8.1.2 Wave-Local Phase Correction

The method of performing the Parareal corrections in Fourier space as outlined in the section above is a global approach at introducing phase correction. For it to work, all waves must move at the same wave-speed everywhere in space.

In this section, we instead derive a modified correction scheme where we attempt to locally adjust the Parareal correction to take into account advective effects. We leave the zero'th iteration to generate U_n^0 from $u_o = u\left(\bar{x}, t=0\right)$ for all $n \leq n_t$ unmodified as the sequential application of $\mathcal{G}_{\Delta T}$. In deriving the modification, we assume the behavior of an advection dominated 1D advection-diffusion equation like (6.1), but without assuming the wave-speed to be constant throughout the domain. The question as to how the update equation itself should be modified comes down to interpretation. Consider the below two ways of writing the Parareal correction (5.10),

$$U_{n+1}^{k+1} = \mathcal{F}_{\Delta T} U_n^k + \left[\mathcal{G}_{\Delta T} U_n^{k+1} - \mathcal{G}_{\Delta T} U_n^k\right] = \mathcal{G}_{\Delta T} U_n^{k+1} + \left[\mathcal{F}_{\Delta T} U_n^k - \mathcal{G}_{\Delta T} U_n^k\right] \tag{8.1}$$

The equations are the same, but their interpretation becomes important when considering how to do phase-correction. Should we locally estimate the difference in amplitude and phase of $\mathcal{G}_{\Delta T} U_n^{k+1}$ and $\mathcal{G}_{\Delta T} U_n^k$, and then add it to $\mathcal{F}_{\Delta T} U_n^k$, or should we locally estimate the difference in amplitude and phase between $\mathcal{F}_{\Delta T} U_n^k$ and $\mathcal{G}_{\Delta T} U_n^k$, and then add it to $\mathcal{G}_{\Delta T} U_n^{k+1}$? The two approaches are not necessarily the same. One might reckon that it makes the most sense to correct $\mathcal{F}_{\Delta T} U_n^k$ using the difference between $\mathcal{G}_{\Delta T} U_n^k$ and $\mathcal{G}_{\Delta T} U_n^{k+1}$, but this is not necessarily so. We may somewhat easily estimate differences between $\mathcal{F}_{\Delta T} U_n^k\left(\bar{x}\right)$ and $\mathcal{G}_{\Delta T} U_n^k\left(\bar{x}\right)$, at any point in space, through Taylor expansion around $\bar{x}$, since they started from the same U_n^k, but estimating the difference in behavior of $\mathcal{G}_{\Delta T} U_n^k$ and $\mathcal{G}_{\Delta T} U_n^{k+1}$ is not so easy because they were applied to different solution states U_n^k and U_n^{k+1}, i.e. the change may be larger than what we can reasonably interpolate from a local expansion.

For the reasons mentioned, we choose the first approach, i.e. correcting upon $\mathcal{G}_{\Delta T} U_n^{k+1}$. Let's estimate the new iterate $U_{n+1}^{k+1}(\bar{x})$ as $\mathcal{G}_{\Delta T} U_n^{k+1}(\bar{x})$ plus some correction due to local amplitude and phase difference between $\mathcal{F}_{\Delta T} U_n^k(\bar{x})$ and $\mathcal{G}_{\Delta T} U_n^k(\bar{x})$. We denote $\alpha_\delta^{\bar{x}}$ as the correction due to local amplitude change, and $\phi_\delta^{\bar{x}}$ the correction due to local wave-speed difference, between $\mathcal{F}_{\Delta T} U_n^k(\bar{x})$ and $\mathcal{G}_{\Delta T} U_n^k(\bar{x})$, so that the modified Parareal correction becomes

$$U_{n+1}^{k+1}(\bar{x}) = \mathcal{G}_{\Delta T} U_n^{k+1}(\bar{x}) + \alpha_\delta^{\bar{x}}\left(\mathcal{F}_{\Delta T} U_n^k, \mathcal{G}_{\Delta T} U_n^k\right) + \phi_\delta^{\bar{x}}\left(\mathcal{F}_{\Delta T} U_n^k, \mathcal{G}_{\Delta T} U_n^k\right) \tag{8.2}$$

The correction $\phi_\delta^{\bar{x}}$ due to phase change may be written as a first order approximation

$$\phi_\delta^{\bar{x}}\left(\mathcal{F}_{\Delta T} U_n^k, \mathcal{G}_{\Delta T} U_n^k\right) = \left[s_{\mathcal{F}^k}^{\bar{x}} - s_{\mathcal{G}^k}^{\bar{x}}\right] \nabla_{\bar{x}} \mathcal{G}_{\Delta T} U_n^{k+1}(\bar{x}) \tag{8.3}$$

where $s_{\mathcal{F}^k}^{\bar{x}}$ and $s_{\mathcal{G}^k}^{\bar{x}}$ are the distance a wave moves in a time-subdomain intervalΔT. I.e, $s_{\mathcal{F}^k}^{\bar{x}}$ is the distance a point $(\bar{x})$ has moved after applying $\mathcal{F}_{\Delta T}$ to U_n^k, similarly $s_{\mathcal{G}^k}^{\bar{x}}$ is the distancea point $(\bar{x})$ has moved after applying $\mathcal{G}_{\Delta T}$ to U_n^k. The correction $\alpha_\delta^{\bar{x}}$ of amplitude can be written as

$$\begin{aligned}\alpha_\delta^{\bar{x}}\left(\mathcal{F}_{\Delta T} U_n^k, \mathcal{G}_{\Delta T} U_n^k\right) = & \mathcal{F}_{\Delta T} U_n^k(\bar{x}) - \mathcal{G}_{\Delta T} U_n^k(\bar{x}) \\ & - \left(s_{\mathcal{F}^k}^{\bar{x}} \nabla_{\bar{x}} \mathcal{F}_{\Delta T} U_n^k(\bar{x}) - s_{\mathcal{G}^k}^{\bar{x}} \nabla_{\bar{x}} \mathcal{G}_{\Delta T} U_n^k(\bar{x})\right)\end{aligned} \tag{8.4}$$

Inserting (8.3) and (8.4) into (8.2), we recover

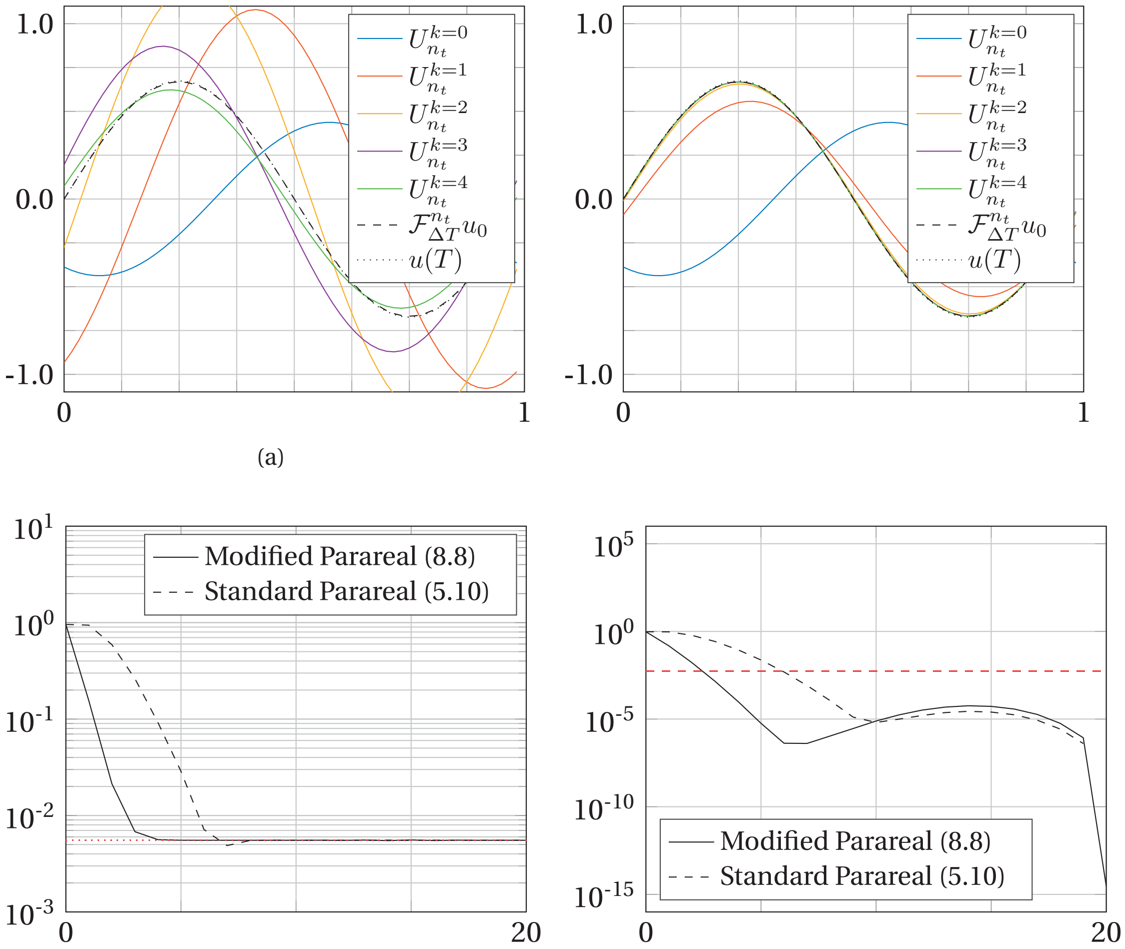
$$\begin{aligned}U_{n+1}^{k+1}(\bar{x}) = & \left(1 + s_{\mathcal{F}^k}^{\bar{x}} - s_{\mathcal{G}^k}^{\bar{x}}\right) \nabla_{\bar{x}} \mathcal{G}_{\Delta T} U_n^{k+1}(\bar{x}) + \left(1 - s_{\mathcal{F}^k}^{\bar{x}} \nabla_{\bar{x}}\right) \mathcal{F}_{\Delta T} U_n^k(\bar{x}) \\ & - \left(1 - s_{\mathcal{G}^k}^{\bar{x}}\right) \nabla_{\bar{x}} \mathcal{G}_{\Delta T} U_n^k(\bar{x})\end{aligned} \tag{8.5}$$

as a modified correction procedure. The question that remains is, how may $s_{\mathcal{F}^k}^{\bar{x}}$ and $s_{\mathcal{G}^k}^{\bar{x}}$ be evaluated? This turns out to be slightly problematic. Even if we knew the exact value of $s_{\mathcal{F}^k}^{\bar{x}}$ and $s_{\mathcal{G}^k}^{\bar{x}}$, they might be so long that the result of the taylor expansion around $(\bar{x})$ is inaccurate. We get around this by recognising that we should be able to estimate $s_{\mathcal{F}^k}^{\bar{x}} - s_{\mathcal{G}^k}^{\bar{x}}$ for any $(\bar{x})$ quite accurately assuming ΔT small, thus we add another equation

$$s_\delta^k(\bar{x}) = s_{\mathcal{F}^k}^{\bar{x}} - s_{\mathcal{G}^k}^{\bar{x}} \tag{8.6}$$

where $s_\delta^k(\bar{x})$ can be approximated quite easily. That leaves us with two equations and three unknowns. To ensure a unique solution, we now demand that correction procedure should yield the same result both forwards and backwards in time. I.e, imagine that $U_{n+1}^{k+1}(\bar{x})$ was known, and that we instead wanted to find $\mathcal{F}_{\Delta T} U_n^k(\bar{x})$ as the unknown. Using the same approach outlined as before, we find that this entails

$$\begin{aligned}\mathcal{F}_{\Delta T} U_n^k(\bar{x}) = {} & \left(1 + s^{\bar{x}}_{\mathcal{F}^k} - s^{\bar{x}}_{\mathcal{G}^k}\right)\nabla_{\bar{x}}\mathcal{G}_{\Delta T}U_n^k(\bar{x}) + \left(1 - s^{\bar{x}}_{\mathcal{F}^k}\nabla_{\bar{x}}\right)U_{n+1}^{k+1}(\bar{x}) \\ & - \left(1 - s^{\bar{x}}_{\mathcal{G}^k}\nabla_{\bar{x}}\right)\mathcal{G}_{\Delta T}U_n^{k+1}(\bar{x})\end{aligned} \tag{8.7}$$

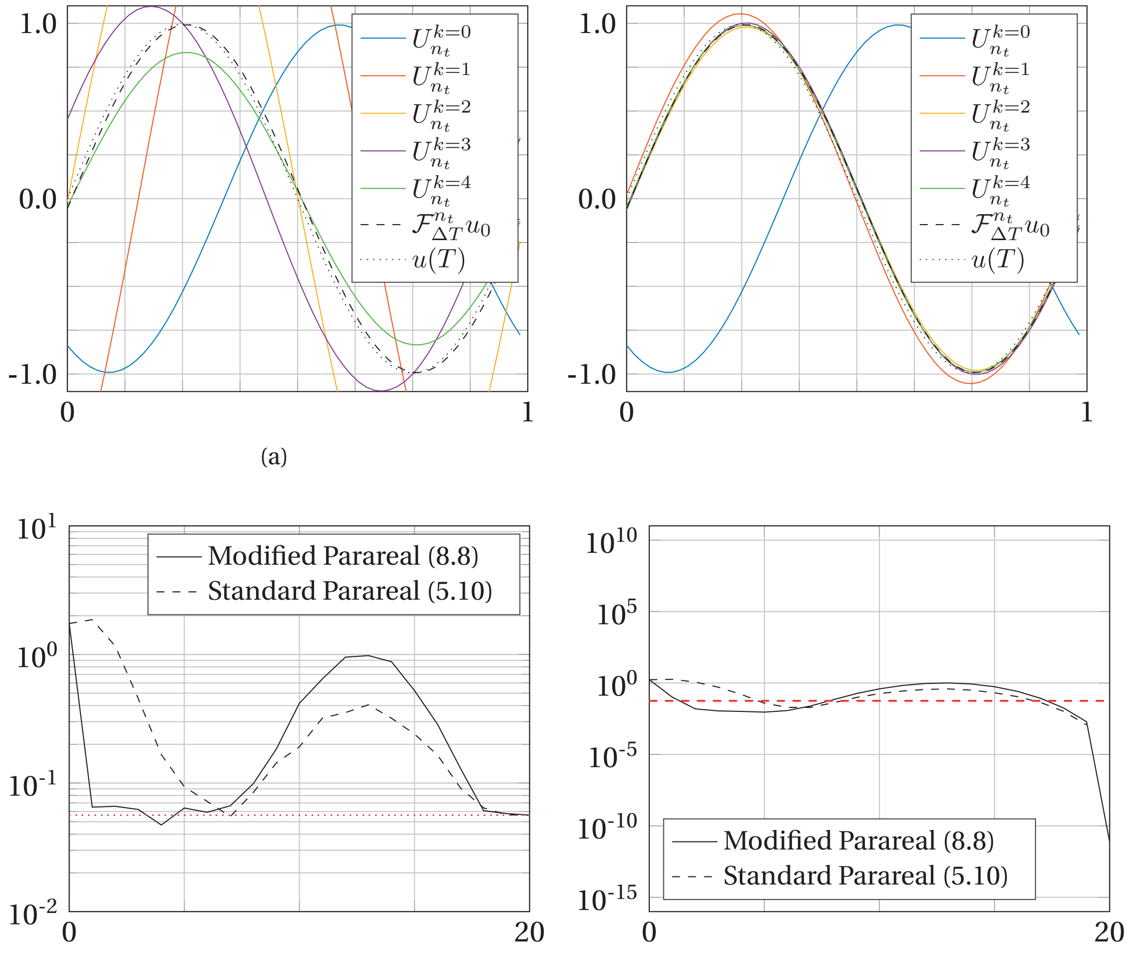


(a)

(c)

(d)

Figure 8.2 – Convergence of the original Parareal method, and the modified variant, when computing an approximation to the solution of (6.1) with $\kappa = 10^{-3}$ and $a = 1$ to $T = 10$ on $n_t = 20$ time-subdomains using a factor 6 coarsened space-time grid as $\mathcal{G}_{\Delta T}$. (a) show iterations at $T = 10$ of the original method (b) shows iterations of the modified variant. (c) error in infinity norm measured with respect to the true solution $u(T = 10)$ and (d) error in infinity norm measured with respect to the sequential fine solution at $u(T = 10)$.



(a)

(c)

(d)

Figure 8.3 – Convergence of the original Parareal method, and the modified variant, when computing an approximation to the solution of (6.1) with $\kappa = 10^{-5}$ and $a = 1$ to $T = 10$ on $n_t = 20$ time-subdomains using a factor 6 coarsened space-time grid as $\mathcal{G}_{\Delta T}$. (a) show iterations at $T = 10$ of the original method (b) shows iterations of the modified variant. (c) error in infinity norm measured with respect to the true solution $u(T = 10)$ and (d) error in infinity norm measured with respect to the sequential fine solution at $u(T = 10)$.

when assuming that $s^{\bar{x}}_{\mathcal{F}^k} = s^{\bar{x}}_{\mathcal{F}^{k+1}}$ and $s^{\bar{x}}_{\mathcal{G}^k} = s^{\bar{x}}_{\mathcal{G}^{k+1}}$. This leaves 3 equations with 3 unknowns from which the modified correction procedure for computing $U^{k+1}_{n+1}(\bar{x})$ is found to be

$$U^{k+1}_{n+1}(\bar{x}) = \mathcal{F}_{\Delta T} U^k_n(\bar{x}) + \left(1 + s^k_\delta(\bar{x}) \nabla_{\bar{x}}\right) \left(\mathcal{G}_{\Delta T} U^{k+1}_n(\bar{x}) - \mathcal{G}_{\Delta T} U^k_n(\bar{x})\right) \tag{8.8}$$

It turns out that this new update equations actually looks a lot like the original Parareal method, with the exception of the multiplication by a factor $1 + s^k_\delta(\bar{x}) \nabla_{\bar{x}}$ on the difference $\mathcal{G}_{\Delta T} U_n^{k+1}(\bar{x}) - \mathcal{G}_{\Delta T} U_n^k(\bar{x})$. Numerical experiments testing the method (8.8) are presented in figures 8.2 and 8.3 for $\kappa = 10^{-3}$ and $\kappa = 10^{-5}$ respectively, in both cases $\alpha = 1$. As before, both $\mathcal{F}_{\Delta T}^{T_n}$ and $\mathcal{G}_{\Delta T}^{T_n}$ are constructed using a fourth ordered compact finite difference stencil[216], and $\mathcal{G}_{\Delta T}^{T_n}$ operates on a space-time mesh coarsened by a factor 6. The experiments essentially mirror those presented in section 6.1, except here $n_t = 20$ time-subdomains were used. In the experiments, the exact $s_\delta^k(\bar{x})$ for a single mode was measured and used as it is constant in the entire domain. In practice $s_\delta^k(\bar{x})$ may be estimated locally as long as ΔT is sufficiently short.

8.2 Deriving Parallel-in-Time Integration Schemes

The Parareal modification introduced in section 8.1.2 improves the initial convergence rate on the simple advection dominated problem, but it does not appear to eliminate the issues of instability. In this section we take another approach. Rather than further attempt to modify the Parareal correction procedure, we here develop a new general method for deriving parallel-in-time integration schemes. We maintain the idea of having a “fine” level operator $\mathcal{F}_{\Delta T}$, for which we wish to find approximations $U^n(x) = \mathcal{F}_{\Delta T}^n u_0$ at all $n < n_t$ time-subdomain interfaces, without allowing for the sequential application of $\mathcal{F}_{\Delta T}$. Inspired by Parareal, we instead allow for one or more coarser levels to be computed sequentially and thereby in some sense act as preconditioners.

In order to present the method for deriving parallel-in-time integration schemes, imagine first that for any given discretization, there exists some operator $\mathcal{H}$ that transforms $\mathcal{F}_{\Delta T} U(x)$ into $\mathcal{G}_{\Delta T} U(x)$ for any solution state $U(x)$, and that the operator depends on some set of functions $c_1(x,t)$, $c_2(x,t)$, ..., so that one may write

$$\mathcal{G}_{\Delta T} U(x) = \mathcal{H}\left(\mathcal{F}_{\Delta T} U(x), c_1(x,t), \ldots\right) \tag{8.9}$$

Can we somehow make an assumption on the behavior of $\mathcal{H}$ with respect to the parameters so to construct a parallel-in-time type scheme? Say for example that in (8.9), $\mathcal{H}$ depends only on a single parameter function $c(x,t)$. If this was the case, maybe we could use $G_{\Delta T} U_n^k(x)$ and $\mathcal{F}_{\Delta T} U_n^k(x)$ to approximate $c(x,t)$. Given $c(x,t)$, we could then compute an approximation to $\mathcal{F}_{\Delta T} U_n^{k+1}(x)$ from $G_{\Delta T} U_n^{k+1}(x)$, i.e. extrapolating from previous iterates to approximate $\mathcal{F}_{\Delta T} U_n^{k+1}(x)$ without actually computing it explicitly. This approximation could then be used as $U_{n+1}^{k+1}(x)$ on which $\mathcal{F}_{\Delta T}$ may be computed in parallel for all n in the subsequent iteration as done in Parareal.

If it so happened that we would need a complicated assumption on $\mathcal{H}$ with multiple function parameters $c_1(x,t),\ c_2(x,t),\ldots$ to properly approximate the relationship between $\mathcal{F}_{\Delta T}$ and $G_{\Delta T}$, then a single equation relating $G_{\Delta T} U_n^k(x)$ and $\mathcal{F}_{\Delta T} U_n^k(x)$ wouldn’t be sufficient to uniquely determine a correction procedure. In that case, one could make the correction into a two-step method by just adding another iteration to the equations like so

$$\begin{aligned} \mathcal{G}_{\Delta T}U_n^{k-1}(x) =& \mathcal{H}\left(\mathcal{F}_{\Delta T}U_n^{k-1}(x), c_1^n(x), \ldots\right) \\ \mathcal{G}_{\Delta T}U_n^{k}(x) =& \mathcal{H}\left(\mathcal{F}_{\Delta T}U_n^{k}(x), c_1^n(x), \ldots\right) \\ \mathcal{G}_{\Delta T}U_n^{k+1}(x) =& \mathcal{H}\left(\mathcal{F}_{\Delta T}U_n^{k+1}(x), c_1^n(x), \ldots\right) \end{aligned} \tag{8.10}$$

With three equations, $\mathcal{H}$ is allowed to have two function parameters. Two-step methods have the unfortunate problem of needing a one-step method to get started. An alternative approach, allowing for more parameters without including many previous iterates, would be to insists on some form of symmetry. One could for example demand that the extrapolation procedure gives the same result when done both forwards and backwards.

$$\begin{aligned} \mathcal{F}_{\Delta T}U_n^{k-1}(x) =& \mathcal{H}\left(\mathcal{G}_{\Delta T}U_n^{k-1}(x), c_1^n(x), \ldots\right) \\ \mathcal{F}_{\Delta T}U_n^{k}(x) =& \mathcal{H}\left(\mathcal{G}_{\Delta T}U_n^{k}(x), c_1^n(x), \ldots\right) \\ \mathcal{F}_{\Delta T}U_n^{k+1}(x) =& \mathcal{H}\left(\mathcal{G}_{\Delta T}U_n^{k+1}(x), c_1^n(x), \ldots\right) \end{aligned} \tag{8.11}$$

Yet another approach could be to add an additional coarse layer $S_{\Delta T}$,

$$\begin{aligned} S_{\Delta T}U_n^{k}(x) =& \mathcal{H}\left(\mathcal{G}_{\Delta T}U_n^{k}(x), c_1^n(x), \ldots\right) \\ \mathcal{G}_{\Delta T}U_n^{k}(x) =& \mathcal{H}\left(\mathcal{F}_{\Delta T}U_n^{k}(x), c_1^n(x), \ldots\right) \\ S_{\Delta T}U_n^{k+1}(x) =& \mathcal{H}\left(\mathcal{G}_{\Delta T}U_n^{k+1}(x), c_1^n(x), \ldots\right) \\ \mathcal{G}_{\Delta T}U_n^{k+1}(x) =& \mathcal{H}\left(\mathcal{F}_{\Delta T}U_n^{k+1}(x), c_1^n(x), \ldots\right) \end{aligned} \tag{8.12}$$

Which would then give four equations to work with, allowing for more parameters and therefore potentially increasing the accuracy with which the relation between the coarse and fine layers may be approximated. Essentially infinite many schemes could be created with this approach. The general idea for constructing schemes is outlined in three steps below

1. Make an assumption on $\mathcal{H}$ with some parameter functions $c_1(x),\ c_2(x), \ldots$ that approximate how $\mathcal{F}_{\Delta T}U$ relates to coarser levels $\mathcal{G}_{\Delta T}U$.
2. Add as many steps and/or as many layers as there are parameter functions in $\mathcal{H}$.
3. Derive an expression for approximating $\mathcal{F}_{\Delta T}U_n^{k+1}(x)$ without computing it explicitly, set as next iterate $U_{n+1}^{k+1}(x)$.

Let's try an derive a simple Parallel-in-Time integration scheme using the method outlined above. To keep things simple, we begin by deriving a one-step method using a single coarse layer. This allows $\mathcal{H}$ to have only a single parameter function $c\left(x,t\right)$ in order for the correction-procedure to be unique. What would be a good assumption on the connection between $\mathcal{G}_{\Delta T}U$ and $\mathcal{F}_{\Delta T}U$? The simplest possible model is to simply add the function parameter, i.e.

$$\mathcal{G}_{\Delta T}U\left(x\right) = \mathcal{F}_{\Delta T}U\left(x\right) + c\left(x,t\right) \tag{8.13}$$

With this assumption, i.e. that there is a constant function $c\left(x,t\right)$ that relates $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$ for all $U\left(x\right)$, we may derive the correction procedure that would lead to convergence in a single iteration by writing out the equations

$$\begin{aligned} \mathcal{G}_{\Delta T}U_n^k\left(x\right) &= \mathcal{H}\left(\mathcal{F}_{\Delta T}U_n^k\left(x\right)\right) = \mathcal{F}_{\Delta T}U_n^k\left(x\right) + c^n\left(x\right) \\ \mathcal{G}_{\Delta T}U_n^{k+1}\left(x\right) &= \mathcal{H}\left(\mathcal{F}_{\Delta T}U_n^{k+1}\left(x\right)\right) = \mathcal{F}_{\Delta T}U_n^{k+1}\left(x\right) + c^n\left(x\right) \end{aligned} \tag{8.14}$$

From which $\mathcal{F}_{\Delta T}U_n^{k+1}\left(x\right)$ is isolated, leading to the following correction procedure

$$U_{n+1}^{k+1}\left(x\right) = \mathcal{G}_{\Delta T}U_n^{k+1}\left(x\right) + \mathcal{F}_{\Delta T}U_n^k\left(x\right) - \mathcal{G}_{\Delta T}U_n^k\left(x\right) \tag{8.15}$$

Which we immediately recognize as Parareal! From this we may infer that Parareal may be considered a form of extrapolation method, at time-subdomain interfaces, $\mathcal{G}_{\Delta T}U_n^k\left(x\right)$, $\mathcal{F}_{\Delta T}U_n^k\left(x\right)$ and $\mathcal{G}_{\Delta T}U_n^{k+1}\left(x\right)$ are used to estimate $\mathcal{F}_{\Delta T}U_n^{k+1}\left(x\right)$ under the simple assumption that there exists some function $c\left(x,t\right)$ for which $\mathcal{G}_{\Delta T}U\left(x\right) \approx \mathcal{F}_{\Delta T}U\left(x\right) + c\left(x,t\right)$. So maybe it is not so surprising that Parareal has limitations given how simple the underlying assumption is. Actually, one might even say that it is quite surprising that it works as well as it does!

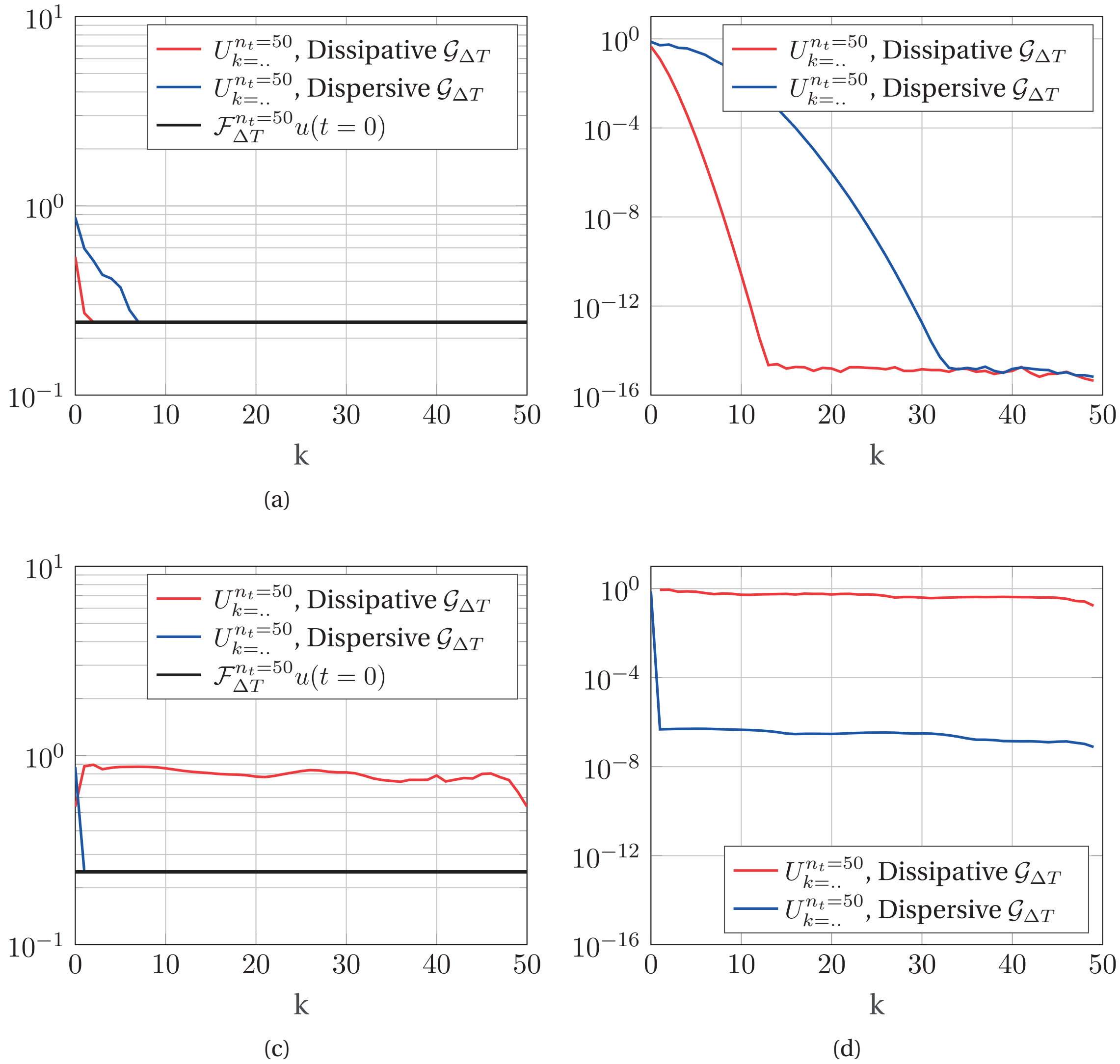
8.2.1 An Exact Correction Procedure

In the previous section we demonstrated how Parareal may be derived with a very simple assumption on the relationship $\mathcal{H}$ between $\mathcal{F}_{\Delta T}$ and a single coarse layer $\mathcal{G}_{\Delta T}$. For certain linear problems, it is possible to write up the exact relation between $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$, so that

$$\mathcal{F}_{\Delta T}U_n^{k+1}\left(x\right) = \mathcal{H}\left(\mathcal{G}_{\Delta T}U_n^{k+1}\left(x\right)\right) \tag{8.16}$$

without the use of any parameters to be approximated by previous iterations $\mathcal{G}_{\Delta T}U_n^k\left(x\right)$ and $\mathcal{F}_{\Delta T}U_n^k\left(x\right)$. This is the case for examples presented in chapter 6 on Parareal applied to the linear advection-diffusion equation. Here we'll attempt to construct an exact $\mathcal{H}$ for the upwind scheme used to solve the linear constant coefficient advection equation with $\alpha = 1$ and CFL $= 0.5$.

We do so for two different coarse operators $\mathcal{G}_{\Delta T}$; One with a dispersive error component with respect to $\mathcal{G}_{\Delta T}$, created using the same discretization as for $\mathcal{F}_{\Delta T}$ but with $\alpha = 1.1$, and one with a dissipative error component, created by adding a diffusion term with $\kappa = 5.0 \cdot 10^{-3}$ to $\mathcal{F}_{\Delta T}$. The error with respect to $\mathcal{F}_{\Delta T}$. is similar in magnitude between the two.



(a)

(c)

(d)

Figure 8.4 – Iterative time-parallel solution of the advection equation, $\alpha = 1$, solved using the upwind scheme with CFL = 0.5 for $\mathcal{F}_{\mathcal{T}}$. The dissipative $\mathcal{G}_{\Delta T}$ was created by adding a diffusion term with $\kappa = 5.0 \cdot 10^{3}$ to $\mathcal{F}_{\mathcal{T}}$. The dispersive $\mathcal{G}_{\Delta T}$ was created by letting $\alpha = 1.05$. In (a)(c), error was measured with respect to the true solution. In (b)(d), error was measured with respect to the sequential fine solution. Parareal was used in figures (a)(b). In figure (c)(d), the correction procedure was performed as described in Section 8.2.1. Integration until $T = 2.5$ on $n_t = 50$ time-subdomains, starting from the discontinuous initial condition depicted in Figure 6.1a. $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ were constructed using 101 points in space with CFL $= 0.5$.

Since the problem and discretization are both linear, the operators $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$ may simple be written as matrix products applied to a vector solution state U_n^k. The exact $\mathcal{H}$ can therefore here be computed by evaluating

$$\mathcal{H} = \mathcal{F}_{\Delta T} \left(\mathcal{G}_{\Delta T}\right)^{-1}. \tag{8.17}$$

The convergence pattern from numerical experiment of using $\mathcal{H}$, as computed above, with the correction (8.16) are presented in figures 8.4 (c)(d). The convergence pattern for Parareal with equivalent $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$ are presented in figures 8.4 (a)(b). As is to be expected, Parareal converges faster when the difference between $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$ is only in terms of dissipation. This is not surprising, and follows the observations presented in chapter 6. What is somewhat surprising is that the reverse is true in our attempt to construct an exact $\mathcal{H}$ for the linear advection equation.

When the error on $\mathcal{G}_{\Delta T}$ is almost exclusively dispersive, the correction procedure (8.16) converges in the first iteration, but when the error on $\mathcal{G}_{\Delta T}$ is dissipative, the correction procedure (8.16) fails. The reason for this behavior is that in order to compute $\mathcal{H}$ from (8.17), one must invert $\mathcal{G}_{\Delta T}$. When $\mathcal{G}_{\Delta T}$ contains dissipative components, this problem becomes ill conditioned. One is essentially trying to rewind a diffusion process, a problem with no unique solution possible solutions, i.e. the matrix to invert becomes close to singular. In the case of having only a little artificial dissipation from the discretization in $\mathcal{G}_{\Delta T}$, the correct solution can be recovered to high accuracy $\sim 10^{-6}$ in a single iteration. When making an assumption on $\mathcal{H}$ containing some parameters, it is thus interesting to note that dissipative effects are both good and bad. On one hand, dissipation appears to help stabilize parallel-in-time schemes of this sort, on the other hand it is difficult to make an accurate extrapolation if there is a lot of dissipation present.

8.2.2 A New 2-Level Scheme

In the beginning of this section we introduced a method for deriving parallel-in-time integration schemes, and showed how the Parareal scheme follows readily from a simple assumption on $\mathcal{H}$. Parareal works poorly when there are differences in dispersive error components between $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$, this is not surprising as the underlying assumption on $\mathcal{H}$, (8.13), that relates $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$, is particularly bad in this case. In this section we will derive a new scheme by making an assumption on $\mathcal{H}$ that is more reasonable when one expects phase-error type differences between $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$. Assuming only phase-error differences, one might speculate that

$$\mathcal{G}_{\Delta T} U\left(x\right) = \mathcal{H}\left(\mathcal{F}_{\Delta T} U\left(x\right)\right) = \mathcal{F}_{\Delta T + c^n(x,t)} U\left(x\right) \tag{8.18}$$

with some set of functions $c^n(x,t)$ would be a better approach. This leads assumption leads us to the equations

$$\begin{aligned}\mathcal{G}_{\Delta T}U_n^k(x) &= \mathcal{F}_{\Delta T+c^n(x)}U_n^k(x)\\ \mathcal{G}_{\Delta T}U_n^{k+1}(x) &= \mathcal{F}_{\Delta T+c^n(x)}U_n^{k+1}(x)\end{aligned} \tag{8.19}$$

In order to isolate $\mathcal{F}_{\Delta T}U_n^{k+1}(x)$, we first do a 1st order Taylor expansion around ΔT to find

$$\begin{aligned}\mathcal{G}_{\Delta T}U_n^k(x) &= \mathcal{F}_{\Delta T}U_n^k(x) + c^n(x)\left.\frac{\partial}{\partial t}\mathcal{F}_{\Delta T}U_n^k(x)\right|_{t_{n+1}}\\ \mathcal{G}_{\Delta T}U_n^{k+1}(x) &= \mathcal{F}_{\Delta T}U_n^{k+1}(x) + c^n(x)\left.\frac{\partial}{\partial t}\mathcal{F}_{\Delta T}U_n^{k+1}(x)\right|_{t_{n+1}}\end{aligned} \tag{8.20}$$

From which $\mathcal{F}_{\Delta T'}U_n^{k+1}(x)$ may be readily isolated to recover the following update equation at time-subdomain interfaces

$$\left[I + c^n(x)\left.\frac{\partial}{\partial t}\right|_{t_{n+1}}\right]U_{n+1}^{k+1}(x) = \mathcal{G}_{\Delta T}U_n^{k+1}(x) \tag{8.21}$$

where

$$c^n(x) = \frac{\mathcal{G}_{\Delta T}U_n^k(x) - \mathcal{F}_{\Delta T}U_n^k(x)}{\frac{\partial}{\partial t}\mathcal{F}_{\Delta T}U_n^k(x)\big|_{t_{n+1}}} \tag{8.22}$$

The notation $\frac{\partial}{\partial t}\big|_{t_{n+1}}$ indicates the partial derivative in time of the solution state taken at t_{n+1}, i.e. the $n+1$'th time-subdomain interface. Normally only the data at t_{n+1} is saved, hence the notation. In order to compute the derivative in time at t_{n+1} numerically, one must therefore save data in some interval around t_{n+1}. This might at first appear somewhat inconvinient, but it turns out that one can avoid having to do so by simply exploiting the fact that the solution state must satisfy the original PDE that we're trying to solve, i.e. by transforming the derivative in time into derivatives in space.

It is worth noting that the correction procedure (8.21) derived above is implicit, i.e. a linear system must be solved to perform the correction. One could however just as well have derived an explicit scheme by instead assuming

$$\begin{aligned}\mathcal{G}_{\Delta T+c^n(x)}U_n^k(x) &= \mathcal{F}_{\Delta T}U_n^k(x)\\ \mathcal{G}_{\Delta T+c^n(x)}U_n^{k+1}(x) &= \mathcal{F}_{\Delta T}U_n^{k+1}(x)\end{aligned} \tag{8.23}$$

With a first order Taylor expansion around ΔT, this would then lead to a scheme on the form

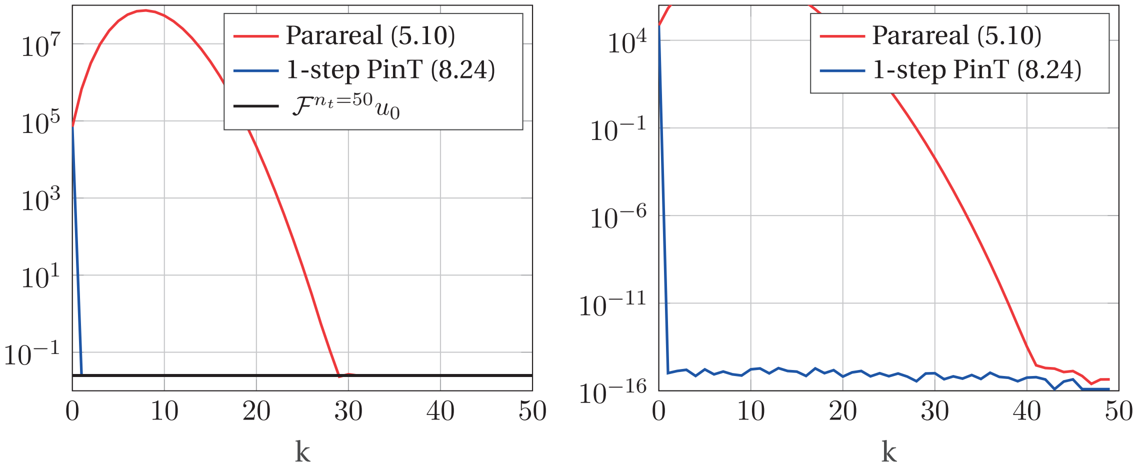
$$U_{n+1}^{k+1}(x) = \left[I + c^n(x) \frac{\partial}{\partial t}\bigg|_{t_{n+1}} \right] \mathcal{G}_{\Delta T} U_n^{k+1}(x) \tag{8.24}$$

with

$$c^n(x) = \frac{\mathcal{F}_{\Delta T} U_n^k(x) - \mathcal{G}_{\Delta T} U_n^k(x)}{\frac{\partial}{\partial t} \mathcal{G}_{\Delta T} U_n^k(x)\big|_{t_{n+1}}} \tag{8.25}$$

which is explicit. Whether one is better than the other is not immediately clear, but it illustrates nicely how many different kinds of method may be derived using the approach outlined earlier. One might speculate that a weighted method between the two would be the best, since the above two methods may be problematic when $\frac{\delta}{\delta t}\mathcal{G}_{\Delta T} U_n^k(x) \approx 0$ and $\frac{\delta}{\delta t}\mathcal{F}_{\Delta T} U_n^k(x) \approx 0$ respectively. Two small test-cases testing the convergence rate of the correction (8.24) is presented in section 8.2.3. In figure 8.5 the method is tested for integration of the scalar ODE test equation, and in figure 8.6 the method is tested for integration of the advection-diffusion equation (6.1) previously used.

In the method just derived we assumed that there was one source of errors between $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$, phase errors. Since we've seen that the assumption (8.13) works well for dissipative differences, we might speculate that a combination of the two would be a good idea, i.e. assuming the relation



(a) (b)

Figure 8.5 – Convergence of two parallel-in-time integration methods when applied to solve the test equation (8.32) as outlined in section 8.2.3. (a) Error measured with respect to the true solution. (b) Error measured with respect to the sequential solution.

$$\mathcal{G}_{\Delta T}U\left(x\right)=\mathcal{H}\left(\mathcal{F}_{\Delta T}U\left(x\right)\right)=\mathcal{F}_{\Delta T+c_1\left(x,t\right)}U\left(x\right)+c_2\left(x,t\right) \tag{8.26}$$

In order to be able to derive a unique update equation for U_{n+1}^{k+1}, we need now two equations. The simplest way of satisfying this is by using an additional previous iteration, i.e. by making a two step method. Writing the equations we arrive at

$$\begin{aligned}
\mathcal{F}_{\Delta T}U_n^{k-1}\left(x\right)=&\mathcal{G}_{\Delta T+c_1^n\left(x\right)}U_n^{k-1}\left(x\right)+c_2^n\left(x\right)\\
\mathcal{F}_{\Delta T}U_n^{k}\left(x\right)=&\mathcal{G}_{\Delta T+c_1^n\left(x\right)}U_n^{k}\left(x\right)+c_2^n\left(x\right)\\
\mathcal{F}_{\Delta T}U_n^{k+1}\left(x\right)=&\mathcal{G}_{\Delta T+c_1^n\left(x\right)}U_n^{k+1}\left(x\right)+c_2^n\left(x\right)
\end{aligned} \tag{8.27}$$

Again we need to do a Taylor expansion around ΔT to be able to isolate $\mathcal{F}_{\Delta T}U_n^{k+1}\left(x\right)$.

$$\begin{aligned}
\mathcal{F}_{\Delta t}U_n^{k-1}\left(x\right)=&\mathcal{G}_{\Delta t}U_n^{k-1}\left(x\right)+\,c_1^n\left(x\right)\frac{\partial}{\partial t}G_{\Delta t}U_n^{k-1}\left(x\right)\Big|_{t_{n+1}}+c_2^n\left(x\right)\\
\mathcal{F}_{\Delta t}U_n^{k}\left(x\right)=&\mathcal{G}_{\Delta t}U_n^{k}\left(x\right)+\,c_1^n\left(x\right)\frac{\partial}{\partial t}G_{\Delta t}U_n^{k}\left(x\right)\Big|_{t_{n+1}}+c_2^n\left(x\right)\\
\mathcal{F}_{\Delta t}U_n^{k+1}\left(x\right)=&\mathcal{G}_{\Delta t}U_n^{k+1}\left(x\right)+\,c_1^n\left(x\right)\frac{\partial}{\partial t}G_{\Delta t}U_n^{k+1}\left(x\right)\Big|_{t_{n+1}}+c_2^n\left(x\right)
\end{aligned} \tag{8.28}$$

From which we derive the correction procedure

$$U_{n+1}^{k+1}\left(x\right)=\mathcal{G}_{\Delta t}U_n^{k+1}\left(x\right)+c_1^n\left(x\right)\frac{\partial}{\partial t}G_{\Delta t}U_n^{k+1}\left(x\right)\Big|_{t_{n+1}}+c_2^n\left(x\right) \tag{8.29}$$

with

$$c_1^n\left(x\right)=\frac{F_{\Delta t}U_n^{k}\left(x\right)-\mathcal{F}_{\Delta t}U_n^{k-1}\left(x\right)-\mathcal{G}_{\Delta t}U_n^{k}\left(x\right)+\mathcal{G}_{\Delta t}U_n^{k-1}\left(x\right)}{\frac{\partial}{\partial t}G_{\Delta t}U_n^{k}\left(x\right)\big|_{t_{n+1}}-\frac{\partial}{\partial t}\mathcal{G}_{\Delta t}U_n^{k-1}\left(x\right)\Big|_{t_{n+1}}} \tag{8.30}$$

and

$$\begin{aligned}
c_2^n\left(x\right)=&\frac{1}{2}\Big(\mathcal{F}_{\Delta t}U_n^{k}\left(x\right)+\mathcal{F}_{\Delta t}U_n^{k-1}\left(x\right)-\mathcal{G}_{\Delta t}U_n^{k}\left(x\right)-G_{\Delta t}U_n^{k-1}\left(x\right)\\
&-c_1^n\left(x\right)\frac{\partial}{\partial t}\mathcal{G}_{\Delta t}U_n^{k}\left(x\right)-c_1^n\left(x\right)\frac{\partial}{\partial t}\mathcal{G}_{\Delta t}U_n^{k-1}\left(x\right)\Big)
\end{aligned} \tag{8.31}$$

The scheme (8.29) is an explicit two-step method. To initiate the algorithm, the original Parareal algorithm, or the method (8.24) just derived, could be used in the first iteration, depending on whether the problem is convection dominated or not. In the section that follows, a few preliminary experiments on the convergence rate of the methods just derived as applied to the scalar ODE test equation, as well as the advection-diffusion equation (6.1), are presented.

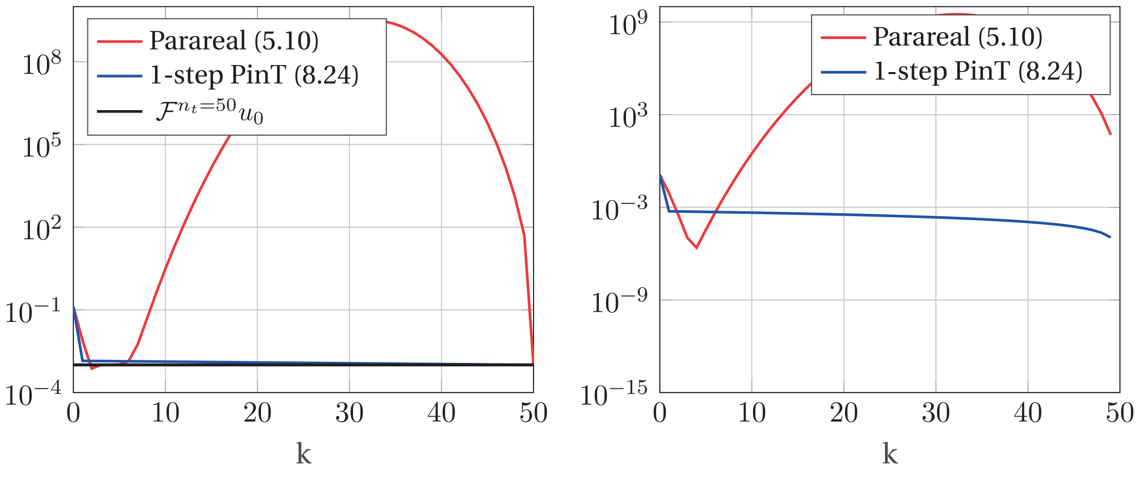
8.2.3 Numerical Experiments

The scalar complex valued test equation is used as a first test for the 1-step parallel-in-time method (8.24)

$$\frac{du(t)}{dt} = \lambda u(t), \quad u(0) = 1, \quad \lambda = -i \tag{8.32}$$

Using $n_t = 50$ time-subdomains of length $\Delta T = 1$ and explicit euler integration for both $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ with timesteps 0.001 and 0.5 respectively. The results are presented in Figure 8.5.

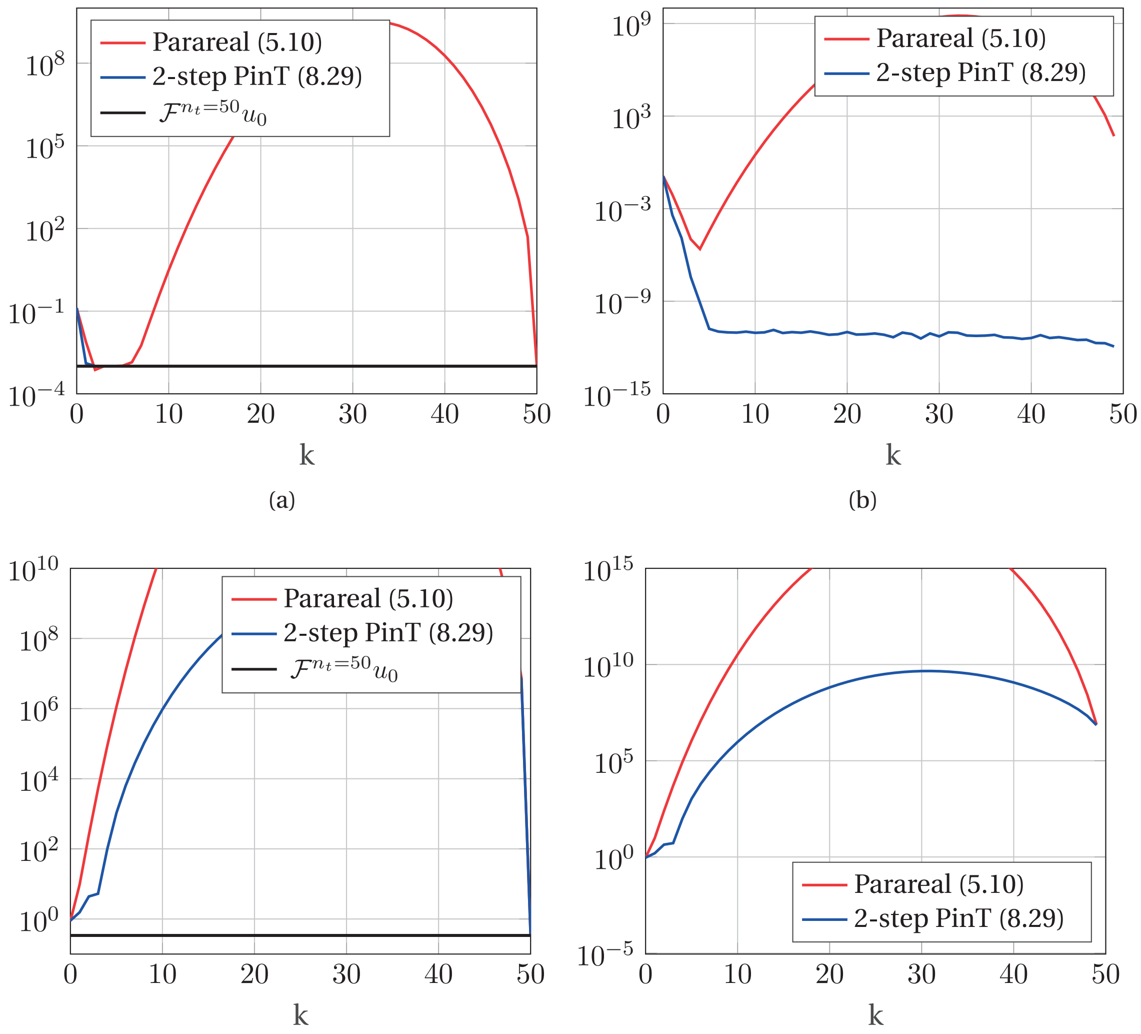
The method converges in a single iteration whereas Parareal diverges. This seems almost too good to be true, but it turns out there's a natural explanation for the extremely fast convergence rate observed in this example. Upon closer inspection one may show that for this particular equation and choice of operators, the assumption on $\mathcal{H}$ after the Taylor expansion is exactly satisfied, and hence convergence in a single iteration is achieved. The example demonstrates that the method introduced here for designing parallel-in-time integration schemes can be used to create schemes that work well on simple cases where Parareal fail spectacularly. Motivated by the promising results, we also tested the method on the advection-diffusion equation (6.1) with $\kappa = 10^{-5}$, $a = 1$ and the smooth initial condition $u(x, 0) = \sin(2\pi x)$. The same fourth order compact finite difference scheme previously used was used for $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ with 201 and 21 points in space respectively.



(a) (b)

Figure 8.6 – Convergence of two parallel-in-time integration methods when applied to solve the advection-diffusion equation (6.1) as outlined in section 8.2.3. (a) Error measured with respect to the true solution. (b) Error measured with respect to the sequential fine solution.

The results are presented in Figure 8.6. Here we note that the new method stagnates. A straight-forward explanation is that the 1-step method (8.24) derived does not correct dissipative errors. This motivated the derivation of the two-step method (8.29). The same advection-diffusion test case was used to test the two-step method, results are presented in Figure 8.7(a)(b) when using a continuous initial condition.



(a) (b)

(c) (d)

Figure 8.7 – Convergence of two parallel-in-time integration methods when applied to solve the advection-diffusion equation (6.1) as outlined in section 8.2.3 with (a)-(b) smooth initial condition, and (c)-(d) discontinuous initial condition. $\kappa = 10^{-5}$, $a = 1$. Integration until $T = 2.5$ on $n_t = 50$ time-subdomains. $\mathcal{F}_{\Delta T}$ 201 points in space and $\mathcal{G}_{\Delta T}$ 21 points in space. (a)(c) Error measured with respect to the true solution. (b)(d) Error measured with respect to the sequential fine solution.

In figures 8.7(c)(d) the two-step method was tested with a discontinuous initial condition as depicted in Figure 6.1a. For the discontinuous case, the two-step scheme derived diverges like Parareal. This indicates that simply increasing the accuracy of the extrapolation is not sufficient to stabilize this approach at creating parallel-in-time integration schemes.

8.3 Summary

The material presented in this chapter are results from work in progress testing recent ideas. For the numerical experiments in section 8.1.2, testing the proposed modification of Parareal, the code-base used for tests in Chapter 6 was reused. In the code, the fourth order discretization for the linear advection-diffusion is hard coded for constant coefficients only. Since the modification should work for non-constant coefficient advection-diffusion as well, investigating this is a first priority in further testing. In addition, it should be tested how the method handles solutions containing higher frequencies.

In section 8.2, a general approach for deriving parallel-in-time integration schemes was derived. Two examples were given and tested. The methods are shown to work very well when only a single mode is present, but like Parareal it too fails for discontinuous solutions. The latter is somewhat disappointing as the two-step scheme corrects on two parameters. It appears that there are still aspects concerning stability that are not understood, so this should be further studied. Ideally, the approach for deriving parallel-in-time integration schemes should be modified in such a way to guarantee stability of methods derived.

Part III

Fault-tolerant Algorithms for Exascale Systems

9 An Introduction to HPC Resilience and Fault-tolerance

Building reliable computing machines has been an ongoing challenge ever since the early days of the first electro-mechanical computers. A design for a general purpose computing machine was published as early as 1837 by Charles Babbage (1791–1871). The *Analytical Engine*, briefly mentioned in Part I, was however never built. Due to its complexity, the estimated cost of construction was so high that no one was willing to fund it[45]. It has since been postulated that despite the design being Turing-complete in principle, the machine, if built, would likely not have worked very well due to limitations on manufacturing technology at the time, likely unable to produce the required reliability of the many components involved. Reliability of operation was likewise a major concern with the first general purpose computer built during the 1940s, and the challenge continues with modern day supercomputers[254].

Computers today achieve reliable operation due to a combination of extreme manufacturing tolerance and accuracy as well as the use of error correcting codes in critical circuits. The reliability of the individual compute nodes that make up modern day supercomputers has become so extreme that if placed in a protected environment, with adequate cooling and a stable power supply, they may operate continuously for several years without failing.

Increased parallelism has become a key element in increasing compute throughput of supercomputers due to fundamental physical laws relating energy consumption and transistor operation as briefly outlined in the introductory chapters. Today, a moderately sized cluster typically operates around 500 compute nodes, and the largest supercomputers in the world $\sim$10.000 nodes. When operating thousands of machines in parallel, even though the mean time between failures (MTBF) of a compute node is measured in years, the time between failure of any single node amongst the thousands of nodes in the machine may be in the order of days or even hours. Limited reliability thus pose a substantial challenge when attempting to run numerical algorithms on the full size of large clusters. The larger the machine, the more likely it is that some component will fail during the computing process. Shrinking processor technology may make processors more susceptible to spurious bitflips due to process variation and manufacturing defects [32, 52], and the crowing complexity of systems will make managing system reliability increasingly difficult [185]. Early studies have estimated that Exascale machines may fail as frequently as once every 30 minutes on exascale platforms [282]. Such failure rates will require new error-handling techniques.

The term *resilience* has thus far been used without an explicit definition. In the thesis we follow the terminology used by Avizienis et al. [12] and adopted in several widely cited papers on the topic of Exascale resilience[53, 282]. We consider the word to be synonymous with *fault tolerance* and define it as in [282]

- "A collection of techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults"

In the literature on fault tolerance and resilience, faults, errors, and failures are sometimes used interchangeably. In the thesis we add a slight distinction to the terms following the definitions in [12].

Fault: The cause of an error (Bug, spurious bit flip, etc.)

Error: The part of total system state that may lead to a failure (Bad values, hardware malfunction, etc.)

Failure: A transition to incorrect service (An event, e.g. no solution, wrong solution, unresponsive nodes, etc.)

The total system state is the set of all states in computation, communication, stored information and interconnects. A fault is said to be active when it cause an error, otherwise it is a dormant fault. Faults are typically local to a single component, this opposed to errors that may potentially propagate trough the machine from one component to another. Errors will not necessarily result in a failure. An extensive overview of challenges in addressing fault-tolerance for future exascale computing systems is given in [282] and [53] along with an overview of sources and potential ways if mitigating the problems. In this chapter, a brief introduction sufficient to supplement the material presented in the subsequent 3 chapters is given.

9.1 Classification of Errors and Failures

In the thesis, two general types of failures are considered, both briefly presented here. Computers may experience faults that lead to an error in registers, memory, storage, or the output of a logic unit which may then manifest itself in the form of one, or more, undetected bit-flips in a word. This may in turn lead to a failure in the form of a wrong solution, polluted by the data corruption. We refer collectively to errors and failures in the form of corrupted program data as Silent Data Corruptions(SDCs).

Spurious bit flips may also lead to other types of errors and failures than an SDC. If the fault occurs in such a way that the resulting SDC leads to the termination of a local MPI process, this will eventually result in the failure of the entire program. Spurious bit-flips are just one of many types of faults that may cause an error in the form of a *permanent* or *transient* compute node loss. Other common faults are hardware malfunctions in the form of failed power-supply units, network cards, etc. Increasingly often, software bugs, rather than hardware malfunction, are determined to be the root cause[99, 206]. Regardless the type of fault, such events will cause abrupt termination of all processes running on the node, likely leading to a failure of the entire application.

In the literature specifically related to HPC on fault tolerance and resilience, SDC type errors and failures are often referred to as *soft*, whereas the loss of compute resources are considered *hard*. Hard errors are sometimes distinguished as being either permanent or transient. In sections 9.1 and 9.1, the impact of soft and hard errors are discussed, and an overview of statistics on their occurrence as available in the literature is presented.

Soft Errors and Failures

Soft errors have the potential to corrupt the solution procedure in ways that may not be immediately obvious to the domain scientist or engineer using the simulations as part of their work. The typically attitude towards SDC resilience is to assume that errors are so rare that they may as well be ignored, favoring the simple solution of doing a re-run if the computational output looks vastly different than what was expected. This approach raises questions of the trustworthiness of numerical simulations performed[54]. In addition it is worth noting that both the cost of an SDC induced re-run and the probability of needing such a re-run scales linearly with the size of the machine.

Soft errors are mostly related to storage elements in the form of spurious bit-flips. A primary source of soft errors arise from energetic particles interacting with the silicon subtract that either flip the state of a storage element or disrupt the operation of a logic circuit. Such events may lead to a silent data corruption (SDC), i.e., no warning or exception is raised but data has been corrupted. Depending on the location of the SDC, it may lead to an event that over the course of many compute cycles turns into a hard error as discussed in section 9.1.

The amount of studies on quantifying the rate at which SDC type errors occur on modern day clusters is somewhat limited in comparison to the work done on developing SDC fault-tolerant algorithms. Despite a sizable amount of research, a consensus on the frequency of SDC type errors seems not yet to have been established[210, 286]. In [286, 285] the authors present a study on soft-errors occurring in the DRAM measuring error rate on ECC type memory, reporting rates of correctable and not correctable errors. A similar study was published in [82] investigating the impact on HPC accelerators. As for CPUs, in [65] the authors present a study on the occurrence of soft-errors when irradiating a Power BQC 16C chip with high-energy particles during execution. They use the measurements to project actual long term failure rate for larger-scale HPC systems. They project using the irradiation experiments the mean time between errors at sea level of the SRAM-based register files and Level-1 caches for a system similar to the scale of Sequoia system with roughly 1.6 million cores to be approximately 1.5 days.

Analyzing DRAM errors based on job scheduler logs and counters in hardware-level error correcting codes has the inconvenience that knowledge about errors escaping hardware checks is incomplete and must be inferred. In [25] a study was presented analyzing memory errors on a 1000-node cluster using low-power memory without error correcting codes. The authors present data from 12000 terabyte hours of error monitoring on 1000 nodes observing 55.000 faults in total. Their study showed that most multi-bit errors corrupted non-adjacent bits in a memory word, and interestingly that most errors flipped memory bits from 1 to 0. The studies confirm that use of error correcting codes in peta-scale systems is essential.

The error correcting codes typically used in HPC use parity bits so to protect words against spurious bit flips typically caused by alpha particle strikes. ECC memory may correct a single bit error and detect any double bit error within a word. The numbers presented also suggest that the extensive hardware based error correction in modern systems is doing a good job making SDCs somewhat infrequent even on Petascale systems. When SDCs do happen, they may have no measurable impact, depending on the location of the fault, as many iterative algorithms used widely used in computational science are inherently soft error resilient [274]. Mitigating the effects of SDC type errors is however still very much an active field of research as it is not clear how well the approach of protecting circuits with error correcting codes will work at exascale. Parity bits and error detection/correction on DRAM is expensive in terms of both cost and energy, and some circuits are not feasible to protect without large overhead. It has been speculated that creating algorithms inherently resilient to SDC type errors may be a cheaper options than increasing hardware protection of vulnerable circuits.

Hard Errors and Failures

Whereas the impact of SDCs on production code remains somewhat speculative, hard errors already present a very real challenge on large compute clusters[282]. Having at least some form of rudimentary fault-tolerance is essential for running code that scales to the full size of peta-scale systems. Simplistic approaches are unlikely to scale exascale systems. Even if the compute nodes in a potential exa-scale system would have an individual MTBF (Mean Time Between Fealure) of a century, a machine with 100.000 such nodes would encounter a failure every 9 hours on average[88]. This being shorter than the execution time of many HPC applications.

Faults that begin as soft-errors but eventually propagate into hard-errors have been a major issue in the past, in particular during the early days of building clusters based on commodity hardware. In a number of notable cases, radiation sources were shown to be the culprit in rendering, at the time being, very large scale clusters useless [131]. On today's clusters, all components from memory to CPUs to networks have some form of error correcting code build in. This does however not mean that soft-errors and SDCs have become non-issues. Some logic units, particularly with-in CPUs are prohibitively expensive to protect with error-correcting code[215]. Typically a hard error involves only a single node, but in particular if critical infrastructure is shared among multiple nodes, such as a power supply units, occasionally hard errors do affect more than one node at a time.

In [146], Gupta et al. presented a study on failures in large-scale systems at Oak Ridge National Laboratory. Their study covered failures of more than one billion compute node hours across five different systems over a period of 8 years. The systems studied varied in size from 0.3 Petalops to 27 Petalops. They found that system software related bugs such as kernel panic and bugs related to the Parallel File System contributed to approximately 20% of failures. This differs somewhat from previous studies indicating that failures due to software bugs is the dominant type of failure[99, 206]. Interconnect technology has improved significantly over the course of the 8 years, and they now constitute a small proportions of errors with hardware malfunction still being the predominant source of failures. Failures due to detected but uncorrected bit-flips, i.e., two or more bit-flips in the same word, are the source of around 15 percent of all failures on Titan GPU based machine. Overall the five systems, normalized to 18688 nodes, has a mean time between failure (MTBF) of 7.5 to 22.7 hours. Any application that wish to scale across the entire machine must therefor be equipped with rudimentary fault-tolerance against hard failures so to avoid wasting compute resources.

The usage of compute accelerators in large scale systems has become common practice, and a number of studies on their reliability has been published [298, 234]. Naturally, adding more components increase the frequency of hard failures. An interesting key point is that although using memory protected with error correcting codes reduce the occurrence of soft errors by several orders of magnitude, they may surprisingly lead to a small increase in the rate of hard failures as otherwise benign double bit corruptions are caught and lead to process termination.

Several studies on the statistics of the occurrence of hard failures among compute nodes in a cluster setting exists. Assuming that hard failures are mutually independent so that failures occur as a Poisson point process, i.e. exponential distribution, is natural and a good first approximation. It has however been demonstrated empirically that failures tend to not be entirely independent in the sense that following a failure the probability of another failure increases. The Weibull distribution with decreasing hazard rate has been shown to model the occurrence of failures well in several studies[269, 319].

9.2 Strategies for Detection and Repair

Some strategies for detection, repair, and recovery are generic, others are algorithm specific, and some are designed specifically to protect against either soft or hard failures whilst others offer protection against both. For soft errors, the main difficulty is the issue of effectively detecting the presence of corrupt data, whereas for hard failures the main issue tends to be dealing with the data lost in a distributed memory application. Some strategies such as process replication are naturally applicable for protection against both soft and hard errors, others such as checkpointing requires the addition of some detection strategy to protect against soft failures. In section 9.2 and 9.2, a brief overview of commonly used methods and recent results is given.

Soft Errors and Failures

Although soft errors are expected to be less frequent than hard errors, they have the added complexity of being difficult to detect. Several recent papers have been published on the topic of understanding how SDC type errors propagate throughout a system and on how to model and predict their impact on programs[47, 191].

A highly robust way of preventing SDC type errors causing soft failures is the use of full or partial replication as has been explored in several studies [114, 36, 28]. An advantage of the replication based approach is that it may be used to protect mitigate the impact of hard failures as well. The disadvantage to the approach is that replication is very costly. One method of reducing the cost is to employ compile time analysis to protect trough replication only statistically vulnerable portions of program code[111, 194].

More algorithm specific cost effective approaches to soft error resilience requires efficient detection of SDC's[20, 35]. Many lightweight detectors has been proposed. The common theme is to allow for some trade-off between detection cost and accuracy. The methods are typically designed to recognize anomalies in HPC datasets based on knowledge on the physical laws or spatial interpolation. Other approaches use various filters and/or time series prediction. Although the methods are not completely accurate, they are able to detect a substantial amount of SDC type errors at a reduced cost [19, 84, 21, 37].

There exists a large body of published research on handling SDC type errors from an application perspective in which fault tolerant versions of commonly used algorithms in computational science has been developed. Methods has been developed for general matrix operations [167, 311], certain iterative solvers [275, 265], sorting algorithms[144], numerical algorithms for PDEs[145, 160] and many others.

Hard Errors and Failures

When creating, or modifying, an application to make it resilient towards hard failure, usually one of three approaches is taken, each of which is outlined in the list below

Backward Recovery: To tolerate hard failures, all current production-quality technologies available rely on checkpointing, also known as backwards recovery[53]. The approach is conceptually simple; write all critical program data to some non-volatile storage at periodic intervals so that, in the event of a fail-stop, the application may be restarted from the most recent checkpoint. This approach has been used since the early days of HPC. It mitigates the impact of hard failures by enabling the application to only have to recompute the progress lost since last checkpoint[189, 88]. The main drawback of checkpointing is the burden it places on I/O infrastructure. Computational power and memory capacity of nodes has increased at faster rate than that of the Parallel-file-system bandwidth, leading to I/O becoming a bottleneck in many applications. In addition, on shared-use distributed memory clusters, checkpointing is likely to interfere with communication and I/O of other applications. The checkpoint-to-filesystem approach works well when using a comparatively small number of nodes, but does not scale well. The mean time between failure scales linearly with the number of nodes used, and so does the associated lost computational work done between the point of failure and the most recent check-point.

Some applications when running on large clusters may need up to several hours to create a checkpoint on parallel file system. This is dangerously close to the MTBF of many machines, current day peta-scale clusters typically have a mean time to failure (MTBF) of any single node of roughly a day[146]. When the time to checkpoint is close to the MTBF of a system, an application may spend all its time in a never ending cycle of checkpoint-restart without making much progress.

Various techniques for reducing the size of checkpoint have been developed or proposed. The proposed methods typically work by exploiting similiaties in data [39], trough data aggregation and compression [173, 74], or trough the use of incremental checkpointing [263, 224]. Other proposed approaches attempt to increase the speed of protection whilst minimizing I/O congestion trough disk-less checkpointing and/or multilevel checkpointing[301, 243, 321, 217]. The idea with multi-level checkpointing is to combine checkpoints to the parallel-file-system with cheaper, and less resilient, checkpoint levels. Light-weight checkpoints can be constructed utilizing node-local storage or memory along with some form of cross-node redundancy or erasure code. If the cheaper checkpoints are able to recover from are comparatively large number of failures, the expensive parallel-file-system checkpoints will not have to be made as often. This leads to higher overall system efficiency as less time is spent creating checkpoint, and recomputing lost work. A more comprehensive introduction to checkpointing technologies is given as part of the introduction to 11.

Forward Recovery: In addition to the problem of protecting data, recovery trough checkpointing has the disadvantage that a lot of compute work is discarded upon rollback. For certain algorithms it is possible to recover without rollback by instead taking extra steps to compensate for the lost data and its effect. This approach is often referred to as algorithmic based fault tolerance (ABFT). Fault-tolerant variants has been developed for the numerical soltuion of certain PDEs[135, 222], for use in N-Body tree computations[56], for the conjugate gradient method[235], and for some general dense matrix operations such as matrix factorization[63, 95].

A requirement for roll-forward recovery is that the application processes and the runtime environment stay alive when experiencing failures. This is not the case with standard MPI, but several resilient MPI designs has been under development, or are currently being developed. Among these, FT-MPI was the first attempt[106], another more recent development is ULFM MPI that allows for notification of errors and add specific functionality to reorganize work by identifying failed ranks and allowing for MPI communicators to be repaired either by removing the failed rank, or by replacing it with a spare rank[40, 7]. Another recent development is the Global View Resilience (GVR) that use versioning distributed arrays for resilience and allow for flexible recovery[66, 67].

Replication: Process replication is another proposed approach to cope with faulty hardware. The idea with replication is to concurrently run one or more replicas of the data and computation so that in the event of failure, no need for forward or backwards recovery is needed[43, 294]. At least two MPI based libraries for process replication to address fail-stop have been developed, rMPI and MR-MPI[112, 102], demonstrating the feasibility. The main issue with repplication is the high resource overhead. A recent study showed that it is possible for repplication to be more effecient than rollback recovery, but only in some extreme situations where the MTBF of the system is extremely low and the time to checkpoint and restart very high[113].

9.3 Contributions in Part III

The remainder of this part of the thesis consists of three chapters, each of which deals with issues of fault-tolerance in HPC. In 10, a modified variant of the Parareal algorithm discussed in great detail in part II is presented. The chapter is an extended version of that published in [231]. The modified algorithm is resilient towards both soft and hard failures, the first such variant of Parareal.

According to a recent study, multi-level checkpointing is likely the most realistic approach for general purpose fault tolerance at exascale[81]. In 11, a new library for multi-level check-pointing is presented along with scaling tests. The library, called Llama, is the first to support both automatic rollback without restart and the use of an arbitrary number of checkpoint levels with topology aware checksum checkpoints of arbitrary group size and number of parity code blocks.

Llama use erasure codes to create resilient memory-conserving checkpoints. Erasure codes are widely used for protecting data in applications spanning everything from Blu-ray Discs to QR codes. A particular important application these days is protecting data in data centers. Erasure codes take a message of k data blocks and transform it into a longer message of n data blocks in such a way that the original message of k blocks may be recovered from some subset of the n blocks. If any subset of k blocks amongst the n is sufficient to recover the original message, the code is said to be optimal. It is in general not possible to recover the original message with less than k blocks, even for optimal codes. In 12, it is demonstrated that, given a little knowledge about the underlying structure of the data encoded, it is possible to partially recover the otherwise lost data even when less than k data blocks are available.

10 Fault-Tolerance in Parareal

In this chapter, a modified version of the Parareal algorithm introduced by Lions et al. in [193] and treated in great detail in Part II, is developed. The modified variant is fault-tolerant in the sense that it is resilient towards the loss of compute nodes and towards the introduction of silent data corruptions, i.e. soft and hard errors. Tolerance towards loss of compute nodes is made possible by using nodes that have already completed their work as spare nodes in a scheme that allows for detection, removal and repair of unresponsive compute resources. Resilience towards silent data corruptions (SDCs) is achieved by viewing the Parareal corrections procedure at time-subdomain interfaces as a series of fixed point methods, each of which is only dependent on previous time-subdomains. Detection of SDC infected iterates is made possible by monitoring the residual, locally at each time-subdomain, for unexpected behavior. The content of the chapter is a slightly extended version of an article published in [231].

10.1 Introduction

Parallel-in-time integration is a technique for extracting additional parallelism in the solution of evolution problems beyond what is possible using standard spacial domain decomposition methods. By introducing a decomposition of the time domain, it is possible, for certain classes of problems, to greatly increase the number of nodes that may be used to accelerate the solution procedure. A space-time parallel algorithm presented in [261] is able to scale to 458,752 cores whilst solving the 3-dimensional heat equation, the full size of the 5 Petaflop/s JUQUEEN cluster. The space-time parallel code scales to three times the number of nodes than the code based on parallel multi-grid alone on the same problem.

In a recent report released by the Exascale Mathematics Working Group at Lawrence Livermore National Laboratory, time-parallel integration techniques are highlighted as a potential path to overcome limitations of strong scaling in evolution problems and calls for more research in the direction [92]. Here research is presented following these directions by showing how Parareal, with slight modifications, may be made resilient towards hardware faults. The chapter begins with a short introduction to time-domain parallelism and the general formalism. In section 10.2 the approach to work distribution and data-locality is outlined along with the strategy and implementation for recovery upon node loss. In section 10.3 we benefit from the interpretation of Parareal as a point-iterative method in the introduction to develop the algorithm resilient to silent data error. The ideas introduced for mitigating the impact of SDC type errors are tested on a parallel-in-time implementation of the solution of an advection-diffusion equation. The final section 10.4 contains a short summary of the chapter.

10.1.1 Parallel-in-Time using Parareal

In section 5.2 of Part II, the Parareal algorithm was presented as a fixed point method. In this chapter, we focus the analysis and implementation od the Parareal algorithm where the preconditioner have the same lower bi-diagonal structure as $M_{\mathcal{F}}$. However, similar fault-tolerant implementations may be constructed for other fixed-point iteration type time-parallel domain-decomposition methods. A recent example of how time-parallel methods may be made resilient to silent data corruption can be found in [143] where it is demonstrated how the Spectral-Deferred-Correction based Parareal algorithm, introduced in [212], may be made resilient by introducing a special strategy for monitoring the residual inside the iterations. A comprehensive introduction to Parareal can be found in [228], and important contributions to the analysis of the method can be found in [288, 14, 128]. The computation of both $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$ and $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$ is likely in itself parallel in some form, and we therefore henceforth apply the generic term "node-group" to refer to whatever combination and number of CPU and co-processor used to apply $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$ to U_n^k. This has the benefit of abstracting away any techniques for parallelism in numerical algorithms potentially used in $\mathcal{F}_{\Delta T}$ and $\mathcal{G}_{\Delta T}$, e.g. conventional spatial domain-decomposition.

10.1.2 Resilience and Fault-tolerance

In [92, 53], resilience to faults is identified as being critical for future exascale HPC systems. The techniques needed to achieve a thousand fold increase in computational capacity, are predicted to also increase the rate of failures on large systems. This poses substantial new challenges in terms of how to effectively use the machines, and on how to assess and assure the correctness of the results of numerical simulations. In the context of parallel integration techniques, the issue of algorithmic resilience is of particular relevance since methods of parallel integration are developed primarily with a focus on extracting parallelism in the solution of PDE’s beyond what is possible using standard domain decomposition techniques, i.e., simulations involving a very large number of compute nodes. An extensive overview of challenges in addressing faults at exascale is given in [282]. In the white paper, faults caused by malfunctioning hardware are placed into two overall categories, soft and hard node errors. A significant source of soft errors arise from energetic particles interacting with the silicon subtract that either flip the state of a storage element or disrupt the operation of a combinational logic circuit. Such events may lead to a silent data corruption (SDC), i.e., no warning or exception is raised but data has been corrupted. Depending on the location of the SDC, it may lead to an event that, over the course of many compute cycles, turns into a hard error. Hard errors are faults that lead to the complete failure of a node. For current parallel applications based on MPI, the approach for dealing with the loss of a process is to terminate all remaining processes and restart the application at nearest check-point. This approach is costly as many modern clusters scale to thousands of nodes, and the I/O cost of a check-point/restart procedure may be prohibitive. Ideally, a local failure should permit local recovery. Unlike hard errors, soft errors have the potential to corrupt computer simulations in ways that may not be immediately obvious to the domain scientist or engineer relying on them as part of their work, and it is worth noting that both the cost of an SDC induced re-run and the probability of needing such a re-run scales linearly with the size of the machine. Hence, the expectation value of the induced cost scales quadratically, and this may not be acceptable on future exascale systems. In this work we seek to develop a variant of the Parareal algorithm for time-parallel integration that is resilient to both soft and hard errors. As presented in [124] some parallel integration methods are intimately related, and we therefore conjecture that our ideas may extend to other techniques of time-parallel integration.

10.2 Recovering from Node-Loss

The Parareal correction (5.10) may be implemented in different ways. The simplest approach is to divide work into two phases; a purely sequential phase, computing $\mathbf{U}_{n+1}^{k+1}$ from $\mathcal{G}_{\Delta T}\mathbf{U}_n^{k+1}$ with the correction (5.10), and a parallel phase where $\mathcal{F}_{\Delta T}\mathbf{U}_n^k$ is computed in parallel on $n \in N$ nodes. Ideally, the wall-time $T_\mathcal{G}$ for a node group to compute $\mathcal{G}_{\Delta T}\mathbf{U}_n^{k+1}$ is much smaller than the wall-time $T_\mathcal{F}$ to compute $\mathcal{F}_{\Delta T}\mathbf{U}_n^k$, and the limiting factor in obtainable speed-up will be the number of iterations $k_{conv} < N$ required for convergence. In practice however, it is rarely possible to construct a coarse operator $\mathcal{G}_{\Delta T}$ that is sufficiently cheap that its cost may be ignored. Fortunately, there exists many other ways to schedule the computational work than relying on two strictly separated phases, switching between the sequential computation of $\mathcal{G}_{\Delta T}\mathbf{U}$ and the parallel computation of $\mathcal{F}_{\Delta T}\mathbf{U}$.

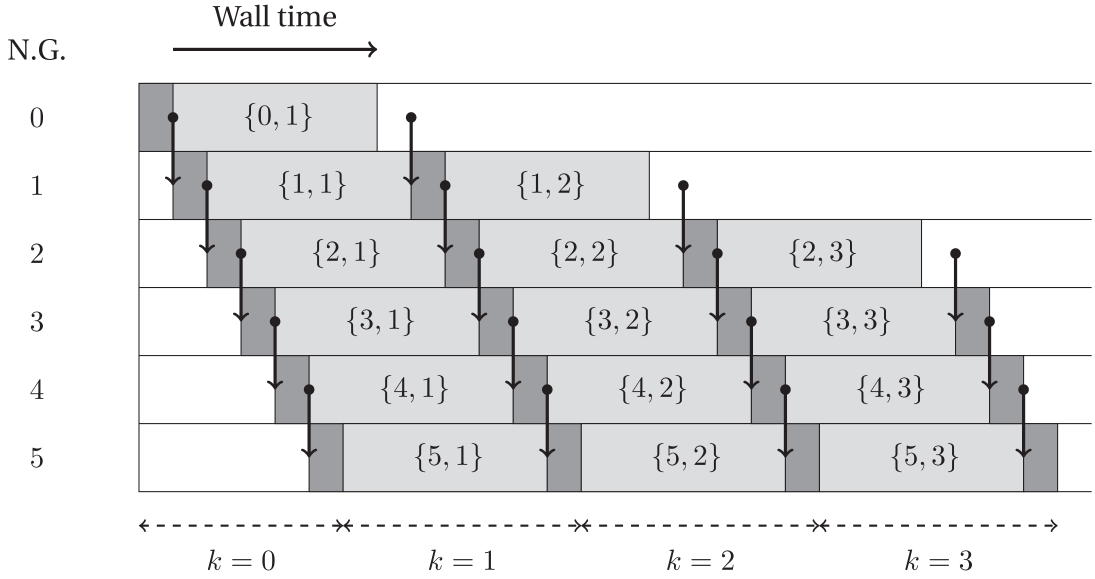


Figure 10.1 – The “Fully-Distributed” work scheduling of the Parareal algorithm as proposed in [9]. Light grey boxes indicate a node-group for computing $\mathcal{F}_{\Delta T}\mathbf{U}$ and dark gray indicates that a node group is evaluating the coarse operator $\mathcal{G}_{\Delta T}\mathbf{U}$. Drawn for $N = 6$ time-subdomains and convergence in $k_{conv} = 3$ iterations.

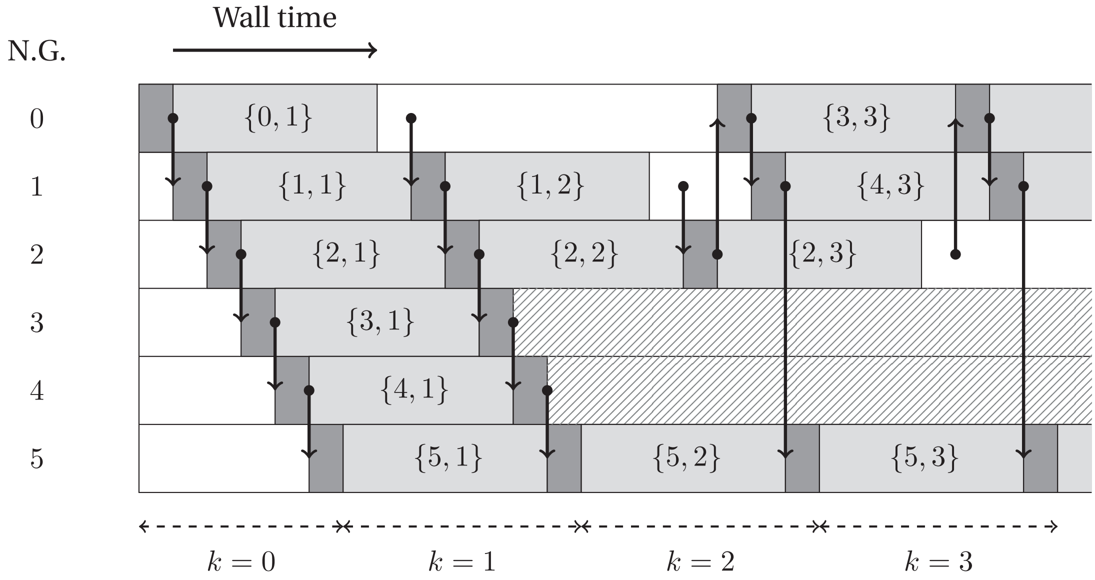


Figure 10.2 – Schematic visualization of the recovery procedure of the Fault-Tolerant algorithm with $N = 6$ time-subdomains and fault-free convergence in $k_{conv} = 3$ iterations. Failed node-groups injected at $\{id_{\Delta T}, k_{err}\} = \{3,2\}, \{4,2\}$. The north east line pattern indicates failed nodes that no longer participate.

For example, $\mathcal{G}_{\Delta T}^{T_0}\mathbf{U}_0^0$ and $\mathcal{F}_{\Delta T}^{T_0}\mathbf{U}_0^0$ may be computed concurrently. By exploiting such in-dependencies, it is possible, to some extent, to mitigate the effects of a relatively expensive coarse operator $\mathcal{G}_{\Delta T}$.

The most widely cited scheduler for the Parareal algorithm was proposed by Aubanel [9]. The "Fully-Distributed" scheduler is near optimal in exploiting independencies, while at the same time being fairly simple to implement. In Figure 10.1, the scheduler is schematically visualized for a small problem, $N = 6$ time sub-domains, and convergence in $k_{conv} = 3$ corrections. Note how, as the first time-subdomain converges, node-groups remain idle while waiting for the rest of the application to finish. In this section a new variant of this scheduler is introduced. The scheduler uses features of the UFLM MPI framework [40] to build a fault tolerant algorithm. Here node-groups that have already completed their work will be used as spares in the event that a still active node-group is lost.

10.2.1 A Fault-Tolerant Scheduler

In Algorithm 7, pseudo code of the proposed Fault-Tolerant Parareal algorithm is presented. The only difference between it and the original "Fully-Distributed" scheme[9] is the introduction of the function calls check_send, check_recv, and spare. The guideline for the recovery strategy for the fault-tolerant implementation is summarized below. In Figure 10.2 the recovery procedure is schematically visualized for a small problem, $N = 6$ time sub-domains, and convergence in $k_{conv} = 3$ corrections with node-groups lost at $\{id_{\Delta T}, k_{err}\} = \{3, 2\}, \{4, 2\}$.

- **Spare Mode.** When $\mathbf{U}_n^{k+1}$ on a time sub-domain handled by a node-group converges, the node-group will become a spare node-group, ready to receive new instructions.

- **Push Strategy.** Recovery upon failure is initiated by *check_send()*. It searches for available spares to continue the work If no spares are available, the application fails globally. If *check_send()* on $id_{\Delta T}$ successfully connects to a spare node-group, it returns a new inter-communicator for sending to the node-group working on $id_{\Delta T} + 1$.

- **Receiving.** A failed *check_recv()* will wait $cT_{\mathcal{G}}$, $c > 1$, for a signal to connect. If no signal appears, it initiates a global failure. Thus, the loss of a node-group is only recoverable if the loss happens during the computation of $\mathcal{F}_{\Delta T}\mathbf{U}$. If a node-group is in the process of completing the correction (5.10) when the group fails, the local loss of a node-group will lead to global failure. If *check_recv()* on $id_{\Delta T}$ successfully connects to a spare node-group, it returns a new inter-communicator for receiving from the node-group working on $id_{\Delta T} - 1$. The converge flag is set zero so that no intervals, following a node-group failure, converges in the current iteration.

- **Convergence.** In the unprotected algorithm, $\mathbf{U}_n^{k+1}$ on the node-group working on the earliest time sub-domain $id_{\Delta T}$, will converge, i.e, during each iteration at least one time sub-domain converges. In the Fault-Tolerant implementation, we require this to be the case as well and $\mathcal{F}_{\Delta T}\mathbf{U}$ must complete on the node-group with the lowest $id_{\Delta T}$ among active node-groups. This condition is naturally enforced by the wait condition on *check_recv()* this simplifies the algorithm while also ensuring that it converges in a maximum of $k_{conv} = N$ iterations as in the unprotected algorithm.

Pseudo code for *check_send*, *check_recv* and *spare_*node is given in Algorithm 6, 8 and 9 respectively. In each algorithm, a number of procedures are outlined. The procedures involve querying node-groups that are actively computing $\mathcal{F}_{\Delta T}\mathbf{U}$ or $\mathcal{G}_{\Delta T}\mathbf{U}$ for information on their current status. That is, retrieving the local values of k and $id_{\Delta T}$ on node-groups without explicit synchronization. The ideal way of achieving this is to use RMA features introduced in the MPI 3.0 standard. However, for our test implementation, this was not possible since ULFM-1.1 is based on OpenMPI 1.7.

Algorithm 6 Pseudo code for **spare()** function

```
if $id_{\Delta T} = N - 1$ then
    $work \leftarrow$ FALSE
    procedure: Send $exit$ to all node-groups.
else
    procedure: Wait for $exit$ or $work$ signal from any node-group $n$.
end if
if $work$ then
    recv_intercomm ← intercomm $n$
    RC ← receive $k$ and $id_{\Delta T}$ on recv_intercomm
    $id_{\Delta T} \leftarrow id_{\Delta T} + 1$
    RC ← receive $converge$ and $U^k_{id_{\Delta T}}$ on recv_intercomm
    check_recv(RC,$converge$)
    $\tilde{U}^k_{id_{\Delta T}+1} \leftarrow \mathcal{G}_{\Delta T} U^k_{id_{\Delta T}}$
    $U^k_{id_{\Delta T}+1} \leftarrow \tilde{U}^k_{id_{\Delta T}+1}$
    Revoke and free send_intercomm
    if Is $id_{\Delta T} + 1$ being processed on any n.-g.? then
        procedure: Find the node-group $n$ processing time-subdomain $id_{\Delta T} + 1$.
        send_intercomm ← intercomm $n$
    else
        check_send(1,$converge$)
    end if
    $K \leftarrow k + 1$
    for $k = K$ to $K_{max}$ do
        procedure: Execute pseudocode Algorithm 1, line 28 to 54.
    end for
end if
```

Algorithm 7 Pseudocode for a Fault Tolerant version of a "fully-distributed" Parareal implementation.

```
$convergeNext \leftarrow$ FALSE
$id_{\Delta T} \leftarrow id_{NG}$
recv_intercomm $\leftarrow$ intercomm $id_{NG} - 1$
send_intercomm $\leftarrow$ intercomm $id_{NG} + 1$
if $id_{\Delta T} = 0$ then
    $\tilde{U}_0^0 \leftarrow y_0, \tilde{U}_1^0 \leftarrow \mathcal{G}_{\Delta T}\tilde{U}_0^0$
    RC $\leftarrow$ send $\tilde{U}_1^0$ on send_intercomm
    check_send(RC,$converge$)
    $\hat{U}_1^0 \leftarrow \mathcal{F}_{\Delta T}\tilde{U}_0^0, U_1^1 \leftarrow \hat{U}_1^0$
    $converge \leftarrow$ TRUE
    RC $\leftarrow$ send $converge$ and $U_1^1$ on send_intercomm
    check_send(RC,$converge$)
    spare                                  ▷ $id_{NG} = 0$ completed $id_{\Delta T} = 0$, now spare
else
    RC $\leftarrow$receive $\tilde{U}_{id_{\Delta T}}^0$ on recv_intercomm
    check_recv(RC,$converge$)
    $\tilde{U}_{id_{\Delta T}+1}^0 \leftarrow \mathcal{G}_{\Delta T}\tilde{U}_{id_{\Delta T}}^0$
    if $id_{\Delta T}! = N - 1$ then
        RC $\leftarrow$send $\tilde{U}_{id_{\Delta T}+1}^0$ on send_intercomm
        check_send(RC,$converge$)
    end if
end if
$U_{id_{\Delta T}}^0 \leftarrow \tilde{U}_{id_{\Delta T}}^0$
for $k = 1$ to $K_{max}$ do
    $\hat{U}_{id_{\Delta T}+1}^{k-1} \leftarrow \mathcal{F}_{\Delta T}U_{id_{\Delta T}}^{k-1}$
    if $convergeNext$ then
        $converge \leftarrow$ TRUE
        $U_{id_{\Delta T}+1}^k \leftarrow \hat{U}_{id_{\Delta T}+1}^{k-1}$
        if $id_{\Delta T}! = N - 1$ then
            RC $\leftarrow$ send $converge$, $U_{id_{\Delta T}+1}^k$ on send_intercomm
            check_send(RC,$converge$)
        end if
        spare                              ▷ enter spare mode
    end if
    RC $\leftarrow$receive $converge$ and $U_{id_{\Delta T}}^k$ on recv_intercomm
    check_recv(RC,$converge$)
    $\tilde{U}_{id_{\Delta T}+1}^k \leftarrow \mathcal{G}_{\Delta T}U_{id_{\Delta T}}^k$
    $U_{id_{\Delta T}+1}^k \leftarrow \tilde{U}_{id_{\Delta T}+1}^k + \hat{U}_{id_{\Delta T}+1}^{k-1} + \tilde{U}_{id_{\Delta T}+1}^{k-1}$
    if $converge$ & $|U_{id_{\Delta T}+1}^k - U_{id_{\Delta T}+1}^{k-1}| > \epsilon$ then
        $converge \leftarrow$ FALSE
        $convergeNext \leftarrow$ TRUE             ▷ converges in $k = k + 1$
    end if
    if $id_{\Delta T}! = N - 1$ then
        RC $\leftarrow$send $converge$, $U_{id_{\Delta T}+1}^k$ on send_intercomm
        check_send(RC,$converge$)
    end if
    if $converge$ then
        spare                              ▷ enter spare mode
    end if
end for
```

Algorithm 8 Pseudocode for **check_send**(RC,$converge$)

```
if RC $! = 0$ then
    Revoke and free send_intercomm
    if $converge$ then
        $converge \leftarrow$ FALSE
        $id_{\Delta T} \leftarrow id_{\Delta T} + 1$
        $\tilde{U}^{k}_{id_{\Delta T}+1} \leftarrow \mathcal{G}_{\Delta T} U^{k}_{id_{\Delta T}}$
        $U^{k}_{id_{\Delta T}+1} \leftarrow \tilde{U}^{k}_{id_{\Delta T}+1}$
        if Is $id_{\Delta T} + 1$ being processed on any n.-g.? then
            procedure: Find the node-hroup $n$ processing time-subdomain $id_{\Delta T} + 1$.
            send_intercomm $\leftarrow$ intercomm $n$
        else
            check_send(1,$converge$)
        end if
        RC $\leftarrow$send $converge$, $U^{k}_{id_{\Delta T}+1}$ on send_intercomm
        check_send(RC,$converge$)
        $k \leftarrow k + 1$
        $\hat{U}^{k-1}_{id_{\Delta T}+1} \leftarrow \mathcal{F}_{\Delta T} U^{k-1}_{id_{\Delta T}}$
        $U^{k}_{id_{\Delta T}+1} \leftarrow \hat{U}^{k-1}_{id_{\Delta T}+1}$
        $converge \leftarrow$ TRUE
        RC $\leftarrow$ send $converge$, $U^{k}_{id_{\Delta T}+1}$ on send_intercomm
        check_send(RC,$converge$)
    else
        if Any node-group in spare-mode? then
            procedure: Find node-group $n$ in spare-mode and send $work$ signal.
            send_intercomm $\leftarrow$ intercomm $n$
            RC $\leftarrow$ send $k$, $id_{\Delta T}$ on send_intercomm
            RC $\leftarrow$ send $converge$, $U^{k}_{id_{\Delta T}+1}$ on send_intercomm
            check_send(RC,$converge$)
        else
            procedure: Send $exit$ to all node-groups.
            exit                                                    ▷ application failure
        end if
    end if
end if
```

Algorithm 9 Pseudocode for the **check_recv**(RC,$converge$) function

```
if RC $! = 0$ then
    procedure: Wait for $exit$ or $work$ signal from any node-group $n$. If wait-time exceeds $\Delta T_{\mathcal{G}}$,
assume failure and abort program.
    if $work$ then
        recv_intercomm $\leftarrow$ intercomm $n$
        RC $\leftarrow$receive $converge$ and $U^{k}_{id_{\Delta T}}$ on recv_intercomm
        check_recv(RC,$converge$)
    end if
end if
```

Instead of using RMA, a solution where all node-groups spawn two Pthreads in the beginning of the application was chosen. One thread for doing the compute work, as outlined in Algorithm 7 and another thread solely for handling signals and returning information on the local node-group's current k and $id_{\Delta T}$ that may be requested by other node-groups. The Fault-Tolerant version must create $\frac{N}{2}(N-1)$ inter-communicators between N intra-communicators at the onset of the algorithm. For the unprotected algorithm even the loss of a single node within a node-group will lead to global failure. No rearranging will be needed, and creating $N-1$ inter-communicators between the intra-communicators of N adjacent node-groups is sufficient. It is not an option to simply create the new inter-communicator during the recovery process since this would require a global synchronization to shrink the global communicator to create a new inter-communicator. In addition to the cost associated with creating the $\left(\frac{N}{2}-1\right)(N-1)$ extra inter-communicators, the Fault-Tolerant algorithm performs a check after each receive and send operation on the intercommunicators. A check must involve an agreement operation across the local intra-communicator to ensure that all send/recv completes successfully. In the section that follows, a small numerical experiment to examine the cost associated with the added operations is presented.

Numerical Experiments

For testing purposes, the proposed Fault-Tolerant variant and the unprotected algorithm are wrapped around the parallel-in-time integration of an ODE system, $\frac{d}{dt}\mathbf{u} = \Lambda\mathbf{u}$ using an implicit Euler integration scheme on the interval $T = [0, 10]$ with $\mathbf{u}_0 = [1, \ldots, 1]$, Λ being a complex valued diagonal matrix, the dimension of which is given by the number of ranks in space. For the numerical experiment 16 ranks in space were used. The computational complexity of this type of problem is very light, so the ratio $r = \frac{T_{\mathcal{F}}}{T_{\mathcal{G}}}$ is controlled by a sleep function rather than the compute capacity of the node. This approach allows for k_{conv}, r and the number of time sub-domains N to be controlled independently. This mimics a general problem, while at the same time allowing to accurately measure the associated costs of creating a large number of inter-communicators and performing agreement operations on the send/recv operations on inter-communicators between time sub-domains. In Figure 10.3, measurements for a problem with $N = 16$ time sub-domains and a ratio $r = 16$ with $T_{\mathcal{G}} = 2s$ and $\mathcal{G}_{\Delta T}^{T_n}$ fine enough that $\left|\mathbf{U}_n^{k+1} - \mathbf{U}_n^k\right| < \epsilon$ after $k_{conv} = 3$ corrections is presented for different error scenarios on a 2x Xeon E5-2643V3 system. Comparing cases (b) and (c), it is clear that the cost associated with a recovery operation is fairly small, and that the cost due to loss of information, possibly forcing the algorithm to make another iteration or two before converging, is an order or two higher. Likewise, the initial added cost of setting up the inter-communicators and threads for handling signaling is comparatively small. The tests were performed on a workstation where all ranks reside in the same memory space and synchronization and communication is therefore cheap. On a large cluster this will not be the case. Given that the experiments were performed using 256 ranks and still reveals a substantial difference in cost, it seems reasonable to expect that upon node loss in a cluster setting, the largest cost factor would be the extra iterations required due to lost information.

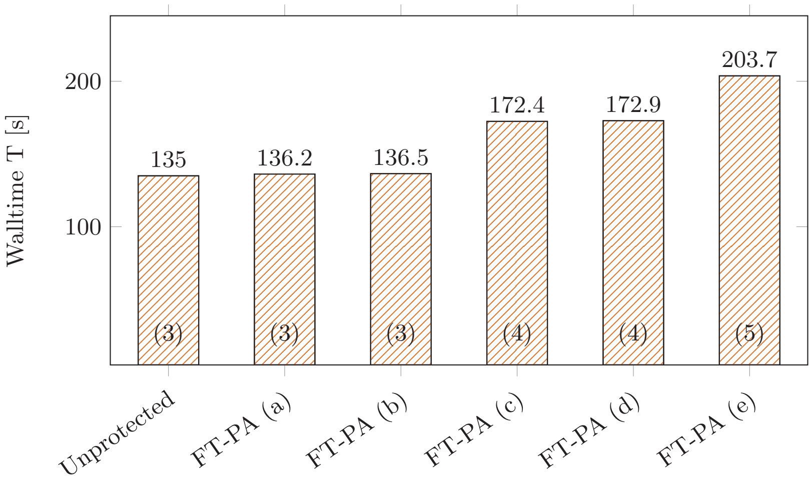


Figure 10.3 – Execution time in seconds for the unprotected algorithm, and for the proposed fault-tolerant algorithm with one or multiple node-group losses located at $\{id_{\Delta T}, k_{err}\}$. (a) No errors. (b) $\{15, 1\}$. (c) $\{15, 3\}$. (d) $\{4, 1\}, \{5, 2\}$. (e) $\{11, 2\}, \{12, 2\}, \{15, 3\}$. The number in parenthesis indicates iterations to convergence. 16 ranks were used in space on 16 time-subdomains for 256 ranks in total.

10.2.2 Failure Analysis

The proposed Fault-Tolerant variant of the Fully-Distributed Parareal work scheduling algorithm may fail to recover when subjected to node-group losses under certain circumstances. As outlined in Section 10.2.1, there is a limit to how many node-groups may be lost at a given iteration, as well as the limitation that all correction operations (5.10), and the computation of $\mathcal{G}_{\Delta T}\mathbf{U}$ must not fail. In this section we derive a lower bound on the probability that the Fault-Tolerant algorithm will execute successfully. A key assumption in our derivation is that the occurrence of node-losses may be assumed to be a Poisson point process, and that statistics on the average time between node failure is available. Let $\mu^{\mathcal{G}}_{\Delta T}$ be the average number of times during a time interval $T_{\mathcal{G}}$ that any node within a node-group fails, and assume that $\mu^{\mathcal{G}}_{\Delta T} \ll 1$. Since the zero'th iteration consists solely of computing $\mathcal{G}_{\Delta T}\mathbf{U}$ N times, each of which must complete correctly, the probability of successfully executing the zero'th iteration is equal to the probability of zero node-losses occurring

$$\mathcal{P}^{N}(0,0) = e^{-N\cdot\mu^{\mathcal{G}}_{\Delta T}} \tag{10.1}$$

For the iterations that follow, the derivation is less trivial. Due to the “push-strategy” for recovery, all subsequent iterations k must have zero node-losses during the correction phase, the probability of which is $\exp\left((k-N)\cdot\mu^{\mathcal{G}}_{\Delta T}\right)$. During the computation of $\mathcal{F}_{\Delta T}\mathbf{U}$, several node-groups may be lost whilst still being recoverable. The probability that n node-groups are lost at iteration k is

$$\frac{\left(r\cdot(N-k)\cdot\mu^{\mathcal{G}}_{\Delta T}\right)^n}{n!}\exp\left(r\cdot(k-N)\cdot\mu^{\mathcal{G}}_{\Delta T}\right) \tag{10.2}$$

where $r=\frac{T_{\mathcal{F}}}{T_{\mathcal{G}}}$. Finally, due to the requirement that the algorithm must converge in a maximum of $k=N$ iterations, $\mathcal{F}_{\Delta T}\mathbf{U}$ on the first active time sub-domain in any iteration k must execute successfully, the probability of which is

$$\exp\left(-r\cdot\mu^{\mathcal{G}}_{\Delta T}\right), \tag{10.3}$$

independent of k. The probability $\mathcal{P}^{N,k}_n$ that the algorithm for N time sub-domains successfully completes iteration k with n node-group loses is then given by the product of (10.2) and (10.3) which becomes

$$\mathcal{P}^N(k,n)=\frac{\left(r\,(N-k)\,\mu^{\mathcal{G}}_{\Delta T}\right)^n}{n!}e^{(1+r)(k-N)\mu^{\mathcal{G}}_{\Delta T}-r\mu^{\mathcal{G}}_{\Delta T}} \tag{10.4}$$

Using the above expression, one may write the probability that the unprotected algorithm executes successfully as

$$P^{N,k_c}_{\text{PA}}=\prod_{i=0}^{k_c}\mathcal{P}^N(i,0), \tag{10.5}$$

as in the unprotected algorithm, each iteration must be completed with $n=0$ node-group losses. In the case of the Fault-Tolerant algorithm it is more complicated to evaluated the probability of successful execution $P^{N,k_c}_{\text{FT-PA}}$. The reason for this being that the number of iterations needed for convergence may increase due to the failures as information is lost, and there are many different potential paths to successful execution.

Figure 10.4 depicts a tree where the branches indicate all possible paths to successful execution for a small example, using only $N=4$ time-subdomains and needing $k_{conv}=2$ iterations to convergence if no failures occur. In drawing the tree it was assumed that for each failure, the number of iterations, needed for convergence, would always increase by one. This will not always be the case however, and the assumption means that evaluating the product of all potential successful completions as drawn in blue circles in the tree would, lead to a lower bound on the combined probability for successful execution. To deduce an expression to describe $P^{N,k_c}_{\text{FT-PA}}$ as a function of all possible N and k_c, we first recognize that at any iteration k, the fault-tolerant algorithm may recover from up to

$$l_1(N,k,n_p)=\min[k-n_p,N-k] \tag{10.6}$$

node-group failures, n_p being the sum of node-group failures in previous iterations $1 \ldots k-1$. The limitation arises due to the need for a spare node-group to be available at node-loss. When a node-group is lost, it may or may not lead to the need for an added iteration before convergence, depending on the location and iteration of the loss. For the purpose of deriving a bound on the probability of the fault tolerant algorithm executing successfully, we assume worst case scenario, i.e. loss of a node-group will always lead to an added iteration, up until the limit that convergence will happen in no less than N iterations. Hence, we define yet another limit

$$l_2\left(N, k_c, n_p\right) = \min\left(k_c + n_p, N\right), \tag{10.7}$$

on the number of iterations needed for convergence. For any given problem, a lower bound on the probability of successful execution may be computed as the sum of the products of each possible path to success. To compute the sum of all possible branches in a tree, as depicted in Figure 10.4, for problems with a large number of time sub-domains, we define a recursive function

$$\Phi^N_{k_c}\left(k, n\right) = \begin{cases} \sum_{i=0}^{l_1(N,k,n)} \mathcal{P}^N\left(k, i\right) \Phi^N_{k_c}\left(k+1, n+i\right) & \text{if } k \leq l_2\left(N, k_c, n\right) \\ 1 & \text{otherwise} \end{cases}. \tag{10.8}$$

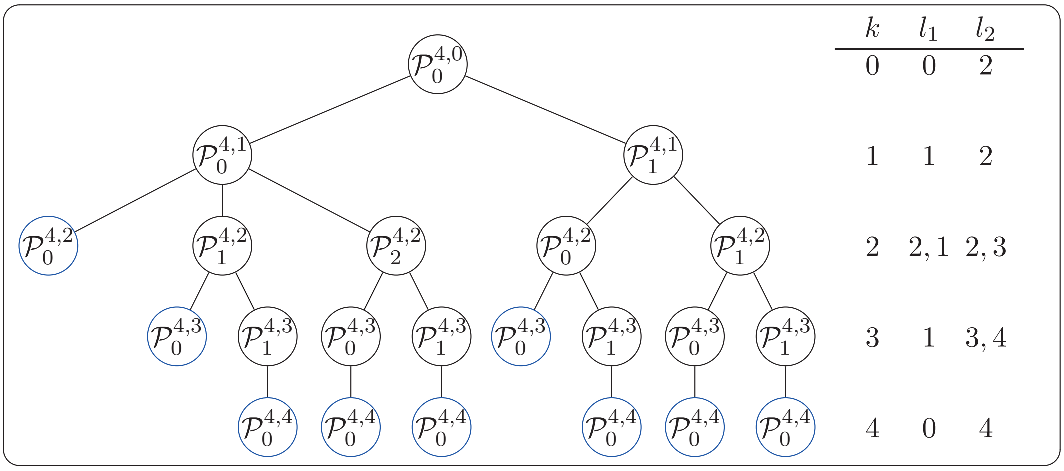


Figure 10.4 – A tree indicating all possible routes to successful execution as drawn for a small problem with $N = 4$ time-subdomains and failure-free convergence in $k_{conv} = 2$ iterations. The probability of successful execution $\mathcal{P}_n^{N,k}$ at a given node may be computed using (10.4). l_1 and l_2 are shown for each iteration k on the right as computed using (10.2) and (10.3). The blue circle indicates successful execution of the algorithm. Branches leading to failed execution are not drawn.

The lower bound on the probability that the fault tolerant algorithm will execute successfully may be written as

$$P_{\text{FT-PA}}^{N,k_c} \geq \mathcal{P}^N \left(0,0\right) \Phi_{k_c}^N \left(1,0\right). \tag{10.9}$$

In Figure 10.5, the probability of successful execution for the unprotected and for the Fault-Tolerant, Algorithm 7, is presented for a problem $N = 16$, $r = 32$ and $k_{conv} = 3$, with on average one node-loss within a node-group per 10000 time-intervals $T_{\mathcal{G}}$, i.e., $\mu_{\Delta T}^{\mathcal{G}} = 0.0001$. In addition, the figure contains a plot of the percentage of failures of the unprotected algorithm that is successfully executed by the Fault-Tolerant algorithm. We denote this ratio R

$$R = \frac{P_{\text{FT-PA}}^{N,k_c} - P_{\text{PA}}^{N,k_c}}{1 - P_{\text{PA}}^{N,k_c}} \tag{10.10}$$

Note that as the number of time-subdomains grows, so does the number of possible paths to success. In the limit $k_c \to N$ and N large, the number of nodes in the tree of possible paths to evaluate approaches the N'th Catalan number. For $N = 28$ that is $\sim 2.6 \cdot 10^{14}$ nodes to evaluate. For large N it is therefore not feasible to evaluate all possible paths to successful execution. We found that a simple alternative to brute force computation of all paths is to prune the tree at some given fixed depth, say 10. By doings so, many theoretically possible, but extremely unlikely paths, are ignored. Since the paths ignored are highly improbable, they contribute little to the overall probability of success. Pruning the tree to traverse does not make the lower bound invalid as removing paths will further decrease the estimate.

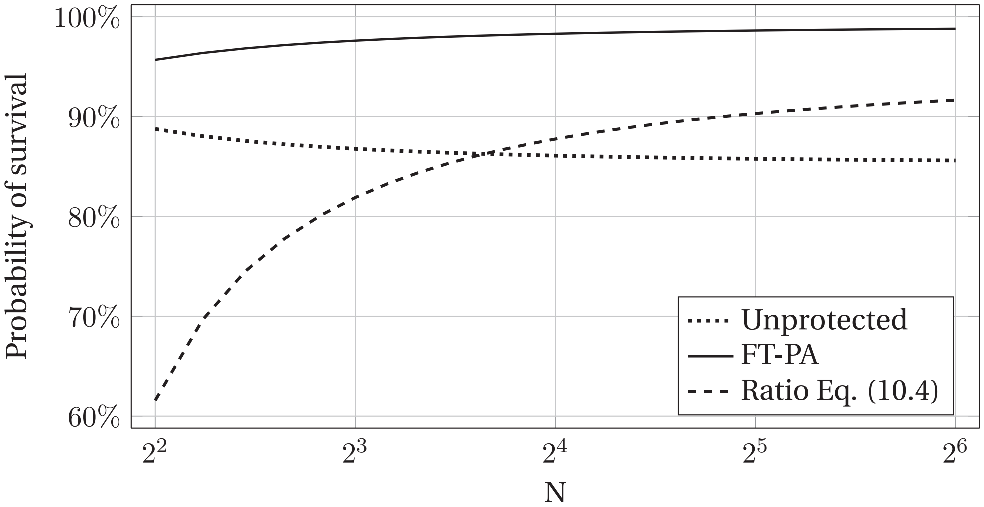


Figure 10.5 – Probability of successful execution for the unprotected algorithm and the Fault-Tolerant variant $N = 16$, $r = 32$, $k_{conv} = 2$ and failure rate $\mu_{\Delta T}^{\mathcal{G}} = 0.0001$. The dashed line indicate the proportion of errors in the unprotected algorithm that the Fault-Tolerant version may recover from.

10.3 Guarding against SDC Errors

In the previous section we approached the issue of node failures. We now consider the other major concern of faults in HPC applications, i.e. soft failures in the form of silent data corruption. When subjected to this type of error, an application may provide an incorrect output without any indication that the application has malfunctioned. Algorithmic resilience towards SDC type errors is an active area of research, and [143] provides a recent example in the context of time integration. In that paper, the authors demonstrate how spectral deferred correction for solving ODE's may be made resilient to SDC type errors and in [31], an auxiliary checking scheme is introduced to form a fairly generic approach to silent error detection in numerical time-stepping schemes. In this work we extend the Parareal algorithm to make SDC resilience an integral part of the algorithm, regardless of the SDC resilience properties of the underlying operators $\mathcal{F}_{\triangle T}$ and $\mathcal{G}_{\triangle T}$.

Numerical algorithms have traditionally been developed under the assumption that all underlying algebraic operations are carried out accurately, subject only to the limitation of machine accuracy. In the following work we stray away from this assumption, i.e., if a matrix vector product results in x, then there is a non-zero probability of computing $x + \tilde{x}$, with $\tilde{x}$ being a random variable. In the field of numerical analysis, a main focus is on the analysis of error and convergence, but since our error is now a random variable, how should we approach the analysis? A natural idea is to define convergence in terms of the statistical moments of the error. This approach was used in [290], where the authors consider a method to be convergent with respect to hardware error, if for every $\epsilon > 0$, a finite amount of work will make $E\left[e\right] < \epsilon$ and $Var\left[e\right] < \epsilon^2$, e being the residual between two consecutive iterations.

10.3.1 SDC Resilient Parareal

In building an SDC resilient algorithm for iterative methods, it is natural to look at the difference between consecutive iterations to detect whether or not an SDC-type error occurred. This is the approach taken by Stoyanov and Webster in [290], where a generic approach for making fixed-point iterative methods resilient towards SDC-type errors is proposed. In that approach, it is argued that if the iteration matrix is a contraction, the norm of the difference between successive iterates should reduce at the same rate as the rate of convergence of the algorithm, thus rejecting iterates if they fail to do so. As presented in the introduction, Parareal is in essence a fixed point iteration, but with a non-normal iteration matrix, the elements of which are potentially non-linear operators. Due to the non-normal structure of the iteration matrix, it is not possible to provide a general guarantee that the iteration matrix will be a contraction. However, since the upper bound on parallel efficiency of the algorithm scales as $1/k_{conv}$, it is reasonable to assume that for any practical application, $\mathcal{G}_{\triangle T}$ is constructed sufficiently close to $\mathcal{F}_{\Delta T}$ so that the iteration matrix will remain a contraction on $\mathbf{U}_n^k$ from $k = 0$ and onwards. Hence, the approach of [290] is applicable to make the Parareal method SDC resilient. However, this approach is limited by the fact that $\bar{\mathbf{U}}^k$ is needed, imposing the need for a synchronization stage between consecutive iterations. This limits the scheduling of work to a slow manager-worker type model. As discussed in Section 10.2.1, such a model is not used in practice as the limitations it imposes on obtainable speed-up are too severe. Fortunately, due to the special structure of the iteration matrix (5.8), we may construct a local approach without the need for synchronization between iterations. First, define the residual between two consecutive iterations on the node-group local time sub-domain n as

$$e_n^{k+1} = \left\| \mathbf{U}_n^{k+1} - \mathbf{U}_n^k \right\|_\infty \tag{10.11}$$

For an SDC resilient model, e_n^{k+1} must be computed at iteration $k+1$ on each time sub-domain n, and communicated along with $converge$, see Algorithm 7, so that the node-group responsible for the n'th time sub-domain at the $k+1$'th iteration can access $e_i^{k+1} \forall i \in 1 \ldots n$. Then, if at any iteration for any time sub-domain

$$\max_{i=0\ldots n} e_n^{k+1} \geq \beta \max_{i=0\ldots n} e_n^k \tag{10.12}$$

is true, we reject $\mathbf{U}_n^{k+1}$ and replace it with

$$\mathbf{U}_n^{k+1} = \mathbf{U}_n^{k-1}, \quad e_n^{k+1} = e_n^{k-1} \tag{10.13}$$

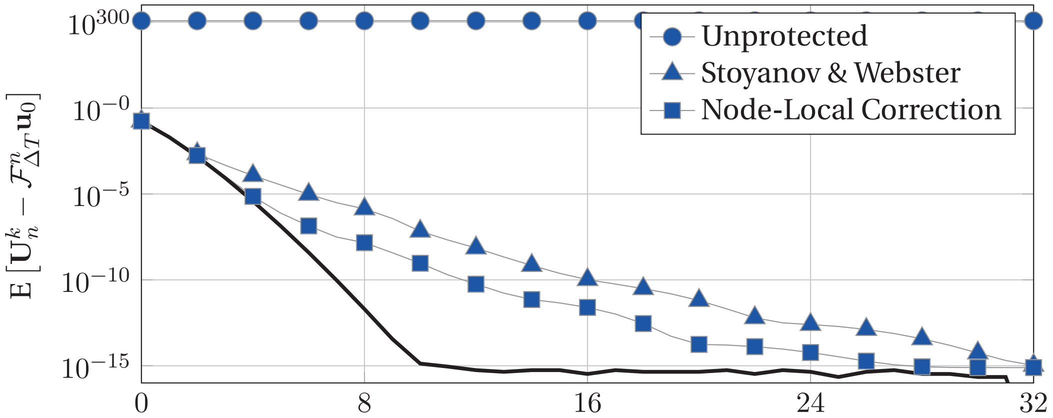
where $\beta \leq 1$ is an upper bound of the contraction factor. If no upper bound is available, using $\beta = 1$ appears to work well. To avoid stagnation due to false rejection, we reject the previous two local iterates. In [290] other approaches for guarding against false rejections are discussed. These approaches only discard a single iterate, but need tunable parameters, or estimates of the Parareal iteration matrix that may not be available in general. In our experience the above approach appears to be near-optimal for avoiding stagnation, while at the same time being parameter-free and easy to implement.

10.3.2 Numerical Experiments

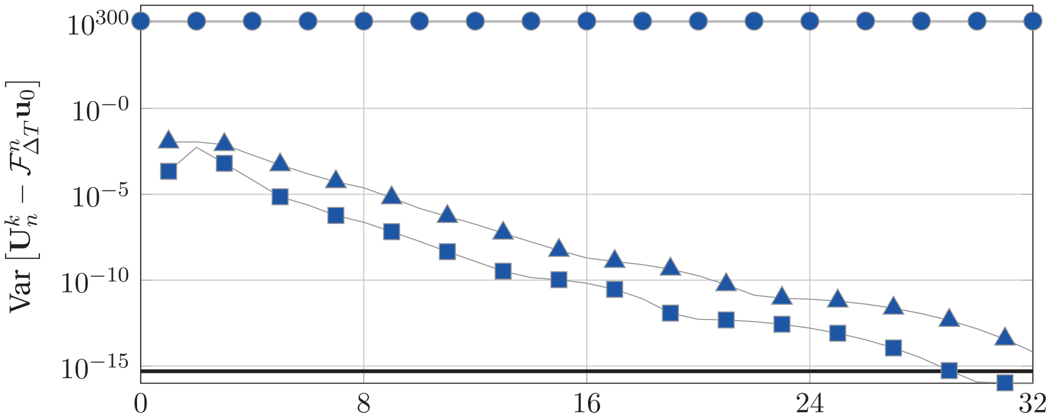
For the numerical experiments, a strategy to introduce data corruption during the solution process is needed. Various studies have attempted to quantify the rate of soft errors leading to SDC's on clusters, see section 9.1 in chapter 9 for a review of studies on the topic. On modern day clusters, DRAM memory and CPU caches are often protected at the architectural level using some type of error correction. SDC type errors may, however, still be caused by external or internal radiation sources, effecting the logic elements within a CPU. It is not at all trivial to deduce how a fault in a logic unit during operation will effect the output of said operation, or it's statistical nature. In [304], a quantitative comparison between the accuracy of direct fault-injection at the assembly code level with that of fault injection in high level code is presented. They demonstrate that faults leading to SDC type errors are well approximated by high level injection of single bit flips. We proceed with this error model for our test-case. At each time-step, in operators $\mathcal{G}_{\Delta T}$ and $\mathcal{F}_{\Delta T}$, every element in the state vector $\mathbf{U}_n^k$ will be subject to a bit-flip with probability P at a random location in the 64bit wide double. As a test-case for SDC-type resilience, we use the time-parallel integration over a wave-period of the 1D advection-diffusion equation

$$\frac{\partial}{\partial t}u(x,t)+\alpha\frac{\partial}{\partial x}u(x,t)=\kappa^2\frac{\partial^2}{\partial x^2}u(x,t) \tag{10.14}$$

with periodic boundaries and advection-diffusion coefficients $\alpha = 1, \kappa = 0.01$.



(a)



(b)

Figure 10.6 – Convergence rate when the solution procedure is subjected to silent-data corruption (SDCs). The unmarked solid black line indicate the convergence rate for the error-free solution procedure. (a) Average error. (b) Error variance.

For the equation, $\mathcal{F}_{\Delta T}$ is constructed using a 4th order compact finite-difference stencil for discretizing the spacial derivatives and $\mathcal{C}^1$-spline collocation for solving the linear system of ODEs[216]. $\mathcal{G}_{\Delta T}$ is constructed using first order finite difference approximations in space and time. We test the solution procedure with a high rate of errors $P = 10^{-6}$, and measure the mean and variance as a function of iterations, averaged over 1000 realizations. We present the results in Figure 10.6. Clearly, the proposed node-local correction strategy is not only preferable in the sense that it is generally applicable regardless of the work-distribution model used. As an added bonus, it also converges faster than the approach proposed by Stoyanov and Webster, this due to the fact that less information is discarded upon rejection.

10.4 Summary

Time-domain parallelism is receiving increasing attention as a viable way to extend the limits of strong scaling in solving evolution-type PDE problems, and offer a potential path to scaling at exascale. We have demonstrated how a novel method of time-domain parallelism, the Parareal method, may be made resilient towards hard errors, when a fault-tolerant supporting API such as ULFM is used. In addition, we have shown that due to the special structure of the iteration matrix, it is possible to monitor the residual between consecutive iterations locally for an SDC resilient correction strategy that may be applied, regardless of the work distribution model used.

The analysis presented in section 10.2.2 may be used to determine if an application should use the regular Parareal scheme or the fault tolerant variant presented. By measuring the overhead related to using the fault-tolerant version, one can a priori for any machine determine if the expected gain in efficiency from running the fault-tolerant algorithm outweighs the loss of efficiency due to its overhead.

A difficulty experienced was the practical aspect of implementing the algorithm using ULFM 1.1. ULFM 1.1 is built on an old version of OpenMPI, with limited support for threading and no RMA. ULFM 2.0 has now been released in sync with the OpenMPI Master. This means better support for threading and the introduction of RMA features introduced in MPI 3. RMA is, however, still not a supported by ULFM. In our experience this limits the extent to which advanced recovery models with asynchronous communication patterns may be efficiently implemented.

11 Llama: A new library for Multi-level Checkpointing

Modern Petascale machines experience node-failures on a regular, sometimes daily, basis[53, 146]. Having to restart a job from scratch is a waste of energy, computational resources, and of the time of the domain scientists using numerical methods as part of their research.

In order to make numerical algorithms scale to the full size of large machines, the de facto standard for fault-tolerance is to have applications periodically write all essential data, required for a restart, to safe storage on a parallel file system[88]. In the event of failure of one or more nodes, the job may simply be rescheduled and restarted from the nearest checkpoint rather than from the beginning.

Periodically creating checkpoints on the parallel file system serves to mitigate the impact of failures on time and resources. The approach has served the community well for many years, but is emerging as a bottleneck for the efficient scaling of algorithms on new machines. For many years, the speed of parallel file systems has been unable to keep up with the increase in compute power and memory of new generations of supercomputers. Time spent by compute nodes on writing data to safe storage is time not spent computing. The impact on parallel efficiency is application dependent, but as much as 10 to 20 percent overhead due to checkpointing is not uncommon[139]. In recent years, new approaches advocating, local recovery for local failures, has started to appear, challenging the notion that compute jobs, spanning hundreds or thousands of nodes, must terminate and restart in the case of failure of a single node. The new approach is necessary to facilitate scaling to big machines of upwards 50.000 nodes such as the planned Exascale machine Aurora to be build at the Argonne Leadership Computing Facility[6]. Even if the mean time between failures (MTBF) of a node is a decade, the machine could experience 10+ node failures per day. Statistics compiled over years of supercomputing usage has shown that the majority of failure incidents involve just a single node, or some small subset of nodes, depending on a shared set of resources. This knowledge may be used to design scalable methods to protect the data needed to rewind an application. In the context of checkpointing, there are many possible ways to protect data locally, or near locally. A simple approach is to assign every rank with a buddy rank, the two may then be made to exchange critical data and keep it locally in-memory. Upon a failure, so long that no two buddies have failed collectively, data at the last point of exchange may be recovered and used for an application restart. Such lightweight local checkpoints may only recover from a subset of possible failure patterns, and it has therefore been proposed to use lightweight checkpoints as a complement to parallel file system checkpoints for increased efficiency and scalability. Protecting data locally is crucial for scalability as it allows to circumvent the bottleneck of the parallel file system. When compute nodes use local storage or memory, the bandwidth for protecting data increases linearly with the number of nodes which serves to avoid limiting network congestion. Several libraries for using lightweight checkpoints, both experimental and for production purposes, has been developed or is currently under development. Approaches of the different libraries under development is discussed in section 11.1.2.

The main topic of this chapter is the introduction of a new Library called Llama. In Switzerland, llamas are used as guards to protect flocks of sheep against predators, lonely or few in numbers. The Llama library uses layers of lightweight checkpoints to guard a cluster of nodes against failures, being particularly effective against local failures of limited scope. Hence the name.

Llama is build on top of ULFM-MPI, Jerasure and GF-Complete and allows for fast and scalable fault-tolerance in time-stepping MPI applications through multi-level checkpointing. A range of checkpoint types are supported, from strong to-disk checkpoints to light-weight in-memory checksum-checkpoints with group-local parallel encoding and decoding for maximum scalability. Light-weight checksum-checkpoints rely on parallel Reed-Solomon encoding and decoding. The library, unlike any other available, supports the usage of an arbitrary number of checkpoint types of arbitrary group and parity size so that checkpoints may be targeted to protect against single node or multiple node failures. It may also be used to provide protection of specific system-resources shared by multiple units, e.g. power supplies and network equipment or potentially entire racks.

The aim of the library is to provide a simple and easy-to-use interface for enabling applications to recover and rewind automatically, without restart, and continue in the face of single and multi-node failures. The library interface and basic usage, along with numerical experiments demonstrating the speed of the library components, is presented in this chapter. The goal is to eventually test the library for protecting applications written in Nektar++, an open-source software framework for high-performance scalable solvers for partial differential equations using the spectral/hp element method[48]. The work presented in this chapter has yet to be submitted for publication.

11.1 Checkpoint-Restart for HPC Fault-Tolerance

An introduction to HPC resilience using checkpointing is given in this section along with a review of libraries currently used, or under development. A brief introduction to User Level Failure Mitigation (ULFM) MPI is presented at the end. ULFM is a proposed extension to MPI, developed by the MPI Forum's Fault Tolerance Working Group. It is not a fault-tolerance library, rather it is an API that allows developers to implement fault-tolerant algorithms in MPI.

11.1.1 Checkpointing for Resilience

In current large-scale distributed memory applications, rudimentary fault tolerance is typically achieved by periodically synchronizing all nodes before writing a solution state to a checkpoint file as first demonstrated in [58]. These checkpoints are written to reliable storage, typically in the form of a parallel file system. Upon failure, an application may restart from a recent state by reading the checkpoint[100, 88]. Synchronization and writing of large files to the parallel file system introduces an overhead, limiting parallel efficiency of applications. Several techniques has been proposed to reduce the size of checkpoints so to minimize the impact. Data may be compressed before being written [74], or one may use an incremental checkpointing approach where only the difference between two checkpoints is stored [263, 224], or combination of techniques including message logging[22, 49]. It has also been proposed to mitigate the impact of to-disk checkpointing by decreasing the frequency of which it is needed, through process replication. By running the same application on 2x nodes, the probability of both failing at the same time decreases substantially, which translates to a decrease in the optimal checkpointing frequency[43, 55, 29].

Diskless checkpointing, obtained by keeping all checkpoint data in-memory, has been recognized early on as one way of accelerating checkpointing[280, 243]. Data may be distributed in such a way that if just a small set of nodes fail, it may be recovered from the nodes unaffiliated with the failure incident. A simple approach to realize this is by simply making many copies of the data to protect and distribute the copies. This approach, however fast, is problematic due to the large memory overhead it incurs. One approach for minimizing the memory footprint of both to-disk and in-memory checkpoint types, is to use RAID techniques or Reed-Solomon based encoding techniques [241, 140]. Creating and storing parity codes has the advantage of requiring less storage, but comes at the cost of needing to encode data to protect it, and decode it upon recovery as well as added implementation complexity. Parity codes may be stored on additional nodes, or may be distributed equally amongst the nodes that created a particular set of parity codes.

In [301], the case was made for using two-level distributed recovery schemes, combining to-disk and in-memory type checkpoints. Their analysis suggested that a two-level system could be substantially more efficient as compared to the use of only a single level. It has been observed that most hard errors only effect a single node and when more nodes fail at the same time they typically do so in a predictable manor, i.e. a power supply unit, serving multiple nodes, fail. Ideally, a local failure should permit local recovery. Multi-level check-pointing addresses this problem by using different types of checkpoints, each of which have their own level of resilience and associated cost. Slower and more resilient levels, such as those made by writing data to the parallel file-system, may allow for recovery from many nodes failures. Cheaper and less resilient checkpoint levels may be utilizing node-local storage or memory[320].

The multi-level approach has received increasing attention from the community in recent years. Machines are now becoming so big, and the discrepancy between compute capacity and speed of parallel file systems so large, that the use of multiple levels of protection is no longer just a question of parallel efficiency and waste of resources. It is speculated that at Exascale, the discrepancy may become so large that some applications will be unable to make computational progress when scaled to the full size of the machine, being trapped in a never ending loop of failure-recovery-restart[282, 88].

Recently, the use of node local solid-state drives and emerging nonvolatile memory technologies has been investigated for possible use in checkpointing[139, 87]. Using local non-volatile storage allow for a fast local approach to protect the data, as with standard random access memory, but without the impact on memory consumption. In [50] it was suggested that checkpointing on node-local SSD drives might even be sufficient for Exascale computation on certain applications.

Frequency of Checkpointing

A key thing to consider is the frequency with which to create checkpoints. Choosing an optimal frequency with which to checkpoint is about balancing the trade-off between loosing as little progress as possible in the event of a rollback, whilst not spending too much time on checkpointing too often. An estimate for the optimal length of the time-interval between checkpoints can be computed using the formula derived by Young [318], which assumes that failures occur as a Poisson process with failure rate λ

$$\tilde{\tau}_{opt} = \sqrt{2T_sT_f} \tag{11.1}$$

where $T_f = 1/\lambda$ is the mean time between failures and T_s the time to protect data. A higher order estimate for the optimal checkpoint interval was derived later by Daly

[79] under the same assumptions

$$\tilde{\tau}_{opt} = \begin{cases} \sqrt{2T_sT_f}\left[1 + \frac{1}{3}\left(\frac{T_s}{2T_f}\right)^{\frac{1}{2}} + \frac{1}{9}\left(\frac{T_s}{2T_f}\right)\right] - T_s & , T_s < 2T_f \\ T_f & , T_s \geq 2T_f \end{cases} \tag{11.2}$$

With regards to multi-level checkpointing schemes, recent studies has investigated methods of optimal selection of the number of levels [30] and the frequency of checkpointing in the different levels[85, 83]. In [293], a general theory for optimal checkpoint placement with arbitrary failure probability distribution was presented and [11] demonstrated how a checkpointing strategy could be modified if some form of online fault-prediction method is available.

Different checkpoint types may have different energy demands and the optimal frequency and number of levels in terms of runtime might therefore not be the same as the optimal frequency and the number of levels in terms of energy consumption, as has been demonstrated and explored in several papers [10, 15, 80]. In most clusters, nodes tend to share certain resources, the failure of which will result in the failure of all nodes. Different such components are likely to have different failure rates. Models to account for this was developed and presented in [13].

Extensive work has been done to develop different models, appropriate under different conditions, for choosing checkpoint levels and frequency of updating checkpoints in the context of optimizing both runtime and energy. One thing all models have in common is that they need estimates of mean time between failure of nodes to compute the optimal frequency of checkpointing. The sensitivity to poor estimates has not been examined as extensively. Studies presented in [178], running simulation using real workload data, suggested that the resulting parallel efficiency of an application is not particularly sensitive to the accuracy of the estimates used. This is perhaps not surprising. If one overestimates the failure rate, this will result in too much time spent checkpointing, but in turn one may save extra time upon recovery and restart. Conversely, if the failure rate is underestimated, an application will spend less time checkpointing. It appears that in practice, the parallel efficiency of a given application is fairly robust with respect to choosing the checkpointing frequency.

11.1.2 Libraries for Automatic Checkpoint-Restart

Several libraries for fault-tolerance in HPC has been developed, or are in the process of being developed, for both experimental and production purposes.

One of the earliest examples of a library for multi-level checkpointing is from 2011, the fault tolerance interface (FTI). In FTI, global parallel-file-system checkpoints is combined with fast checkpoints written to node local storage in the form of SSDs[24].

Along with the checkpoint copy data, Reed-Solomon parity code blocks are computed and written so that an application may be restarted from the lightweight checkpoint even if some failures are non transient, i.e., if a node experienced a hard failure and is no longer available. The authors demonstrated an eight percent checkpointing overhead on the TSUBAME2.0 supercomputer while running at over 0.1 petaflops and checkpointing every 6 minutes. FTI source code is still available in the authors git repository[23].

Scalable Checkpoint/Restart (SCR) is another library for checkpointing. It is well tested and has been used in early versions in production code at LLNL since 2009 and is supported with a comprehensive manual[220]. SCR uses a two-level checkpoint scheme. The more resilient level is a complete check-point to the parallel file system whereas the cheap, less resilient, checkpoint level is constructed using smaller groups of processors that save a check-point locally, either in-memory or on solid-state drives, whilst applying some redundancy scheme across the processors in the group. Two redundancy schemes are supported; a partner/buddy scheme where data copies are distributed, and an XOR model where parity blocks are coded in groups of 8 nodes and distributed evenly within nodes in a group as RAID5. The authors of SCR find that on the systems at LLNL, roughly 85 percent of all node failures may be recovered using the cheaper local checkpoint[217]. SCR may be used in conjunction with ULFM MPI trough the CRAFT library[273]. For a code to use the SCR Library, a list of 11 criteria must be satisfied. Most criteria are trivially satisfied by many applications, except for two

- The code must take globally-coordinated checkpoints written primarily as a file per process.
- On some systems, checkpoints are cached in RAM disk. This restricts usage of SCR on those machines to applications whose memory footprint leaves sufficient room to store checkpoint file data in memory simultaneously with the running application.

The first requirement that a globally-coordinated checkpoint must be written as a file per MPI rank, this may be a limiting factor in some codes. If a code is MPI only, having a separate file for each process per checkpoint will create a very large number of files which may conflict with file system quotas. The second requirement is a potentially limiting factor for applications that are memory bound. Even if the XOR redundancy scheme is used, the total memory footprint of the application will increase by more than a factor of two if solid-state drives are not available.

In addition to FTI and SCR, a library called Fenix has been developed for checkpointing purposes by researchers at Rutgers University and Sandia National Laboratory[121, 122]. Fenix is based on ULFM MPI and provides an API to implement automatic rollback within the application code to avoid restarting jobs that have failed. In Fenix, data is protected by copying the checkpoint in-memory to a partner node's memory. The Fenix library was the first to demonstrate online multi-node recovery and automatic rollback at large scale when injecting actual failures. The library source code is available at a github repository along with rudimentary documentation on usage[296].

11.1.3 User Level Failure Mitigation MPI

The Fenix library achives online rollback, i.e. no restart, by using ULFM-MPI. User Level Failure Mitigation (ULFM) is a proposed extension to MPI developed by the MPI Forum's Fault Tolerance Working Group[40]. It is not a fault-tolerance library, but rather it is an API that allows developers to implement fault-tolerant algorithms in MPI. According to the authors, ULFM was designed to manage failures following three fundamental concepts: 1) simplicity, the API should be easy to understand and use in most common scenarios; 2) flexibility, the API should allow varied fault tolerant models to be built as external libraries and; 3) absence of deadlock, no MPI call (point-to-point or collective) can block indefinitely after a failure, but must either succeed or raise an MPI error. A prototype of ULFM is available to be used with the OpenMPI compiler[42]. As with Fenix, Llama is using ULFM MPI to implement the failure detection, communicator repair, and rollback methods.

11.2 The Llama Library

The Llama library is written in C++ and designed for use with MPI applications. It supports fault-tolerance trough checkpointing with automatic rollback without application restart trough the use of ULFM MPI. To protect application data, the user may specify an arbitrary number of checkpoint levels. The levels may consists of both regular to-disk checkpoints to the parallel-file-system, and topology aware in-memory memory-conserving checksum checkpoints, each of which may be of arbitrary group size with an arbitrary number of checksums parity code blocks. In section 11.2.1, the design of the library is outlined, and in section 11.2.2 the usage of the core functionality is described. Code examples demonstrating usage are given in appendix E.

11.2.1 Design

Llama is designed to be minimally invasive. The target code itself needs very little, if any, modification. Unlike other libraries, rather than modifying preexistent lines of code, code is added to indicate what data is essential to protect and to express how the application should proceed in the event of a rollback. The fundamental object that any application, using the library, must instantiate is the Llama Guard. The object must be created in the beginning of the application and is responsible for keeping track of arrays being protected, and checkpoints used, as well as initiating recovery and repair of the MPI communicator.

The functionality provided by the library is to be used in a loop within which the computational parallel work to protect takes place. Within the loop, the Llama Guard must perform a status check at the beginning each iteration. If the status check detects one or more failed ranks, the communicator will be repaired by the guard by injecting spare-ranks to take the place of failed ranks. The status check call will then return a flag, indicating that the application should perform a rollback and recovery of data. Llama provides an interface trough the Guard that expose data protected at a previous checkpoint to enable a rollback, however it is up to the user to write the functionality needed to do use the data to recover an earlier state as such an operation is application dependent. Instructions on rollback and data recovery is given in section 11.2.2.

Following declaration of the Guard, before entering loop, the library user may add as many levels of checkpoints as needed. Two different categories of checkpoints are supported. The to-disk checkpoint, which protects data by writing it to the parallel-file-system using MPI-IO, and in-memory checksum checkpoints that protects data by encoding parity blocks and distributing these in-memory across ranks. Upon failure, the Guard will find the most recently updated checkpoint for rollback to. If more than one checkpoint has been updated at the same index, the Guard will choose the checkpoint which it perceived as being faster to recover from.

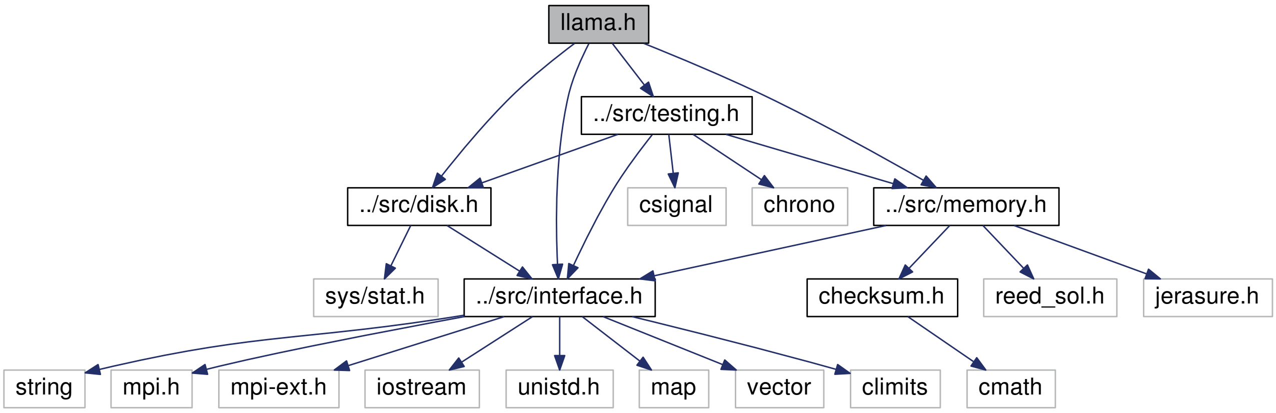


Figure 11.1 – Dependency graph of the Llama header that must be included in applications using the library. The class declaration for the two checkpoint types supported are written disk.h and memory.h. They both derive from a pure virtual class *checkpointinterface* in the interface header. The interface header also includes the decleration of the Guard class. The testing header includes functionality used in the ctest package which includes 30 tests, testing to verify correct execution of the various components of Llama and their interactions.

The implementation of both checkpoint types are derived from a pure virtual class *checkpointinterface*. Other checkpoint types may be added to the library if needed, but to be compatible with the Llama Guard, they must inherit from *checkpointinterface*. Figure 11.1 contains a dependency graph of the Llama header that applications must include. The interface header contains the declaration of both the *Guard* class and the *checkpointinterface* as well as the definition of all templated methods of the two classes. The two methods of checkpointing are explained in greater detail in the two sections that follow.

Protecting data using the parallel file system

MPI-IO is used to write and read checkpoint data to the parallel file system. Distributed 1D arrays are written to a single binary file with each rank writing to it's own section of the file for optimal performance. 2D arrays are protected as arrays of 1D arrays, without closing and opening file objects in between row change. The bandwidth of writing and reading data may be slow for 2D arrays if the number of columns is very small due to the overhead of sending many small messages. This could be improved by implementing features to rearrange data before sending if the number of columns is small, but as of now this has not been implemented. 3D arrays are treated as arrays of 2D arrays, thus they are subject to the same limitation that the data transfer rate may decrease when the number of columns is small.

Protection data using in-memory checksum codes

Protecting data locally in-memory is fast. Transferring data to a nearby node over the network is in general much faster than writing it to a parallel file system. In addition, when compute nodes keep checkpoint data in nearby node-local storage or memory, the bandwidth for protecting data increases linearly with the number of nodes. This makes in-memory checkpointing potentially highly scalable. The disadvantage of protecting data in-memory on nearby ranks is the added consumption of memory which may be a limiting factor in some applications. When protecting data by simply copying it to nearby nodes, the overhead in terms of memory consumption is at least 200 percent.

One way of decreasing the memory footprint, required for a checkpoint, is to use some form of erasure code to encode checksum parity blocks rather than making extra copies of the data to protect. Generally speaking, an erasure code transforms a data block of k symbols into a larger data block of $n > k$ symbols in such a way that the original data block may be recovered using any subset of k symbols from the n symbol data block. Erasure codes are widely used in data-centers to protect data against disk failures.

In the context of HPC, several studies have demonstrated the usage of erasure codes for fault tolerance. In FTI, Reed-Solomon erasure code was used to encode/decode data saved locally on solid state disks[24] and in the SCR library XOR parity bits are computed and stored distributed in-memory to protect against single node failures [217]. In the Llama library, Reed-Solomon encoding is used to create in-memory memory conserving checksum checkpoints. Reed-Solomon codes have found use for a wide range of applications such as storage media, data transmission and data centers[249, 305]. Reed-Solomon codes can be used to encode an arbitrary number of parity codes $p = n - k$ for a data block with an arbitrary number of symbols k thereby allowing for recovery of protected data for up to p failures.

The process of computing p parity code blocks $\{c_1, \cdots, c_p\}$ from k data blocks $\{c_1, \cdots, c_p\}$, can be seen as matrix-vector multiplication where all components are elements in a Galois field

$$\begin{bmatrix} c_1 \\ \vdots \\ c_p \end{bmatrix} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{p,1} & \cdots & a_{p,k} \end{bmatrix} \begin{bmatrix} d_1 \\ \vdots \\ \vdots \\ d_k \end{bmatrix} \tag{11.3}$$

and the matrix A a Vandermonde matrix. In the event that one or more data and/or code blocks are lost, they may be recovered trough another matrix-vector multiplication using the inverse of A, or the inverse of a subset of A along with the remaining code and data blocks.

In Llama, a library called Jerassure is used to create the matrix of Gallois field coefficients, needed for both the encoding and the decoding procedure. Jerasure is an open source library written in C that supports erasure coding in storage applications[244]. Another library, GF-Complete, is used for fast multiplication of coefficients with entire data blocks[242]. The multiplication of each coefficient can be done in parallel as the data blocks are stored on different nodes.

MPI reduce is used to compute the bitwise XOR reduction operation required, to compute the sum for each code-block. Through the parallel reduction operation, the sum is simultaneously computed and distributed. The in-memory checksum checkpoints are created in independent groups of size n, with the p parity codes distributed equally among the n nodes. Equal distribution is achieved trough block level stripping as is done in RAID5 and RAID6, but here implemented in-memory using MPI. With the distribution of parity code-blocks, the memory overhead associated with the checkpoint in addition to the copied data is

$$\sim \frac{p}{n-p}. \tag{11.4}$$

In the limit that the communication in the MPI bitwise reduction operation is the dominant cost of encoding the checkpoint, the time-to-encode will scale as

$$\sim p \lceil \log_2 (n) \tag{11.5}$$

if all ranks in a group are involved in each reduction. In Llama, however, an array of communicators of size np is created when the checkpoint is declared. The array includes all possible patterns of reduction to avoid including group members not relevant in a reduction. The cost of updating the checkpoint therefore scales as

$$\sim p \lceil \log_2 (n - p) \tag{11.6}$$

The improved scaling comes at the cost of an added overhead when creating the checkpoint object as well as when repairing it after a failure. The in-memory checkpoints may be made topology aware in the sense that the user may specify a critical distance that ranks within a group must be separated. If a checkpoint is to protect entire nodes, the user should provide the number of ranks running per node. If the checkpoint is to protect some other entity or multiple nodes, this should be specified in this distance.

2D arrays are protected by treating columns as blocks and each row as a new stripe, similar to RAID6. Due to this design, as with the to-disk checkpoints, performance may be suffer if the number of columns in an array is very small. This may be mitigated by repacking data before protection. 1D arrays are treated by creating a dummy array of pointers to transform it into a 2D array. 3D arrays are treated as many 2D arrays. As with the to-disk checkpoints, declaration is templated so that arrays of all fundamental types may be protected.

11.2.2 Usage

Llama seeks to provide a simple and easy-to-use interface for enabling applications to recover automatically and continue in the face of single and multi-node failures, or the failure of larger shared system resources such as power supplies or network equipment. In this section, library usage is briefly outline. The library uses components of ULFM-MPI, Jerasure and GF-Complete. The latter must therefore be installed first to build the library. After the installation of dependent libraries, Llama can be build from source by cloning the git repository available at [230] with a c4science account. As of now, the build system is rudimentary and the main cmake files needs to be modified manually with the local path to headers and library objects of ULFM-MPI, Jerasure and GF-Complete. Additional details for building the library, and associated examples, are available in the readme file in the repository.

Llama Guards

To make a time-stepping MPI application fault tolerant using Llama, the first step is to create a llama::guard object in the beginning of the application right after *MPI_Initialize*. The constructor for the llama::guard takes three arguments. The first is the address of the global communicator, the second is the number of ranks in the communicator that should be used as spares rather than active workers, and the third a boolean that indicates if the application should throw an error object if recovery is not possible.

Listing 11.1 – Declaration of guard

```
llama::guard my_llama( MPI_Comm* my_world_comm, uint32_t
  my_num_spare_ranks, bool return_errors );
```

If *return_errors* is true, calls to llama::guard methods will throw an error object if recovery is not possible rather than exiting with failure and error message on the std::cerr stream. The situation of recovery not being possible may happen in one of two different situations: A failure pattern for which the checkpoints do not protect has occurred, or if the Guard has run out of spares to replace failed ranks. The Guard, when created, removes *my_num_spare_ranks* number of ranks from the *my_world_comm* communicator, and places an error-handler on *my_world_comm* so that if a rank in *my_world_comm* fails, MPI operation using *my_world_comm* will throw an error object and return rather than abort. In addition to splitting the main global communicator into two pools, a worker and a spare rank pool, and creating an error-handler for *my_world_comm*, the Guard is responsible for the following tasks:

- Keeping track of which arrays need protection
- Keeping track of the status of all checkpoints.
- Monitoring the status of the application and initiating recovery if a failure is detected.
- Choosing which checkpoint to recover from, when a failure is detected.

To notify the llama::guard as to what data needs to be protected, a member function called *ProtectArray* is used.

Listing 11.2 – Method for adding array to guard

```
my_llama.ProtectArray<type T>(const std::string &key, T* p_array,
  uint64_t dim1);
```

Each array to be protected must be supplied with a unique key in the form of an std::string. This key is used to identify which array to recover if recovery is needed.

If one attempts to protect an array with a key that is already in use, ProtectArray will compare the new array and *dim1* input with the already existent. If they are identical, ProtectArray will ignore the call and if not, it will throw an error. *p_array* should be a pointer to the array that needs to be protected and *dim1* the number of elements in the array. Equivalent methods exist for 2D and 3D arrays, with the interface for these are as follows

Listing 11.3 – Method for adding array to guard

```
my_llama.ProtectArray<type T>(const std::string &key, T** p_array,
  uint64_t dim1, uint64_t dim2);
my_llama.ProtectArray<type T>(const std::string &key, T*** p_array,
  uint64_t dim1, uint64_t dim2, uint64_t dim3);
```

Arrays that contain vital information to allow for a quick restart of the application should all be marked for protection. The arrays may be marked for protection in the beginning of an application right after declaring the llama::guard, but may also be added dynamically during the time-stepping phase. Be aware, though, that ProtectArray is collective over all workers, i.e., all ranks in *my_world_comm* after creation of *my_llama*. Each rank must therefore specify an array to be protected, it is not required that the arrays are of the same size for all ranks. Though performance and memory efficiency is higher if all arrays belonging to the same key is of similar size. Similarly, there is an overhead involved in creating and updating arrays. The 1D arrays should ideally exceed 1000kb for the overhead involved in the synchronization between ranks to be mostly negligible. If a given array no longer needs to be protected, the llama::guard can be notified through the methods

Listing 11.4 – Method for removing array from guard

```
my_llama.ForgetArray(const std::string &key);
my_llama.ForgetAllArray();
```

Checkpoint Objects

Under the hood, arrays are protected by checkpoints. The user must specify which kind of checkpoints should be used. Llama supports two overall types of checkpoints; checkpoints to the parallel-file-system and checkpoints that are kept in-memory and distributed in groups across ranks. The constructor for creating checkpoints of the first type looks like this

Listing 11.5 – Declaration of to-disk type checkpoint

```
llama::checkpoint::disk my_disk_checkpoint(llama::guard* my_llama);
llama::checkpoint::disk my_disk_checkpoint(llama::guard* my_llama,
  std::string file_name );
```

```
llama::checkpoint::disk my_disk_checkpoint(llama::guard* my_llama,
  std::string file_name, std::string relative_path );
```

The first argument is a pointer to the llama::guard used, here simply *my_llama*. The second argument is a string, specifying the file name to use for the file written to the parallel-file-system to protect the data. If no name is supplied, a default name convention is followed. The third argument is a relative path in case the user wishes that the checkpoint files are placed somewhere else than in the same folder as the executable. Checkpoints written to the disk has the advantage of being completely resilient, i.e. all arrays marked for protection may be recovered no matter how many ranks are lost. The only limitation to recovery is the size of the pool of spare ranks. The disadvantage of checkpoints on the parallel-file-system is that they are slow to create for large jobs using hundreds of nodes as the bandwidth is limited. Llama supports light-weight in-memory checksum checkpoints. Checkpoints of this type are stored in-memory along with a small parity code that can be used to recompute the data lost if ranks have failed. Parity codes are computed locally in groups in parallel, which allows for the bandwidth of data encoded to scale linearly with the number of ranks in a job. Reed-Solomon codes are used for a limited memory footprint with high resilience. An in-memory checkpoint may be created like this

Listing 11.6 – Declaration of in-memory type checkpoint

```
llama::checkpoint::memory my_memory_checkpoint(llama::guard* my_llama
  , uint32_t connected_ranks, uint32_t num_ranks_per_group, uint32_t
  num_parities_per_group);
```

As with the to-disk checkpoint, the first argument is a pointer to the llama::guard which serves to notify the Guard of the existence of the checkpoint and as to give the checkpoint access to information about which arrays must be protected. The second argument indicates the number of consecutive ranks that is deemed likely to fail together. This would most often be a node-entity, i.e. the number of ranks spawned on each node, but could in practice be anything. For instance, if multiple nodes share a power supply, then a checkpoint could be created with *connected_ranks* being the number of ranks which depend on the same power-supply. The third argument *num_ranks_per_group* is the size of the groups in which the parity data will be computed. The parity data is stored distributed across all ranks within a group. The fourth and final argument is the number of parity data blocks to compute within a group. In simpler terms, *num_parities_per_group* indicates how many failures are allowed to occur simultaneously within a group.

Say for example that *num_parities_per_group* = 1 and *num_ranks_per_group* = 8, then in each group of eight sets of *connected_ranks*, all protected arrays are recoverable so long as no more than one set of *connected_ranks* have failed. The group size *num_ranks_per_group* must be chosen so that *num_worker_ranks* / (*num_ranks_per_group*

* *connected_ranks*) is a positive integer. The choice of *num_ranks_per_group* and *num_parities_per_group* will not only affect how resilient to failures the checkpoint is, but also how fast the checkpoint can encode and decode the parity data for the protected arrays as well as its memory footprint. In general, as *num_ranks_per_group* becomes larger, the memory footprint will become smaller but the encoding and decoding time will become larger. As *num_parities_per_group* increase, the checkpoint becomes more resilient since more sets of *connected_ranks* may fail without compromising the protected arrays, but the memory footprint increases and so does the encoding and decoding time. A simple constructor also exists to create a fast in-memory checkpoint layer

Listing 11.7 – Declaration of in-memory type checkpoint

```
llama::checkpoint::memory my_memory_checkpoint(llama::guard* my_llama
  , uint32_t connected_ranks);
```

Here we only notify the checkpoint of the number of *connected_ranks* that are expected to be likely to fail together. The checkpoint sets *num_ranks_per_group* will then be set by the llama::guard to be the integer nearest to eight that satisfies *num_worker_ranks* / (*num_ranks_per_group* * *connected_ranks*) == integer. *num_parities_per_group* will be set to one. We consider a group size of eight with one parity block to be a good lightweight checkpoint for most purposes, thus it is the default option. With a group size of eight, the checkpoint is fairly memory conserving without being slow to encode or decode and one parity block will allow for recovery upon failure of up to one set of *connected_ranks* in every eighth set. When a checkpoint has been declared, it may be updated by the user using the *Update()* method.

Listing 11.8 – Method for updating checkpoint data protection

```
my_disk_checkpoint.Update(uint64_t idx);
my_memory_checkpoint.Update(uint64_t idx);
```

The update method will loop over all arrays that has been marked in the llama::guard as critical and protect the data in each of them. The to-disk type checkpoints simply write the array to the parallel-file-system whereas the in-memory checkpoints will create an in-memory copy as well as encode parity data to be able to decode and recover data if ranks are lost. The provided index is essential. Upon recovery, the llama::guard will choose the checkpoint with the highest index among the checkpoints that are able to recover from a given failure pattern. Any checkpoint can be updated at anytime during the time-stepping procedure that the user deems necessary. If several different recoverable checkpoints with the same idx exist, the llama::guard will recover the data content using the checkpoint it deems to allow the fastest recovery.

Checking Application Status and Recovering Data

Here we outline how to recover the application in the event that failure on one or multiple ranks occur during a time-stepping procedure. A member method exists

Listing 11.9 – Method for Guard to check application status

```
my_llama.CheckStatus(uint64_t* idx)
```

CheckStatus() plays an important role in the fault-tolerent time-stepping procedure and in the recovery. All spare ranks that were split from *my_world_comm* will proceed until they reach the *CheckStatus()* call and wait here until needed to replace a failed worker rank. In the beginning of each step in the time-stepping loop, the status of the application should be checked. The call to *CheckStatus()* will return one of four different statuses

- LLAMA_STATUS_SUCCESS
- LLAMA_STATUS_REPAIRED
- LLAMA_STATUS_FAILED
- LLAMA_STATUS_COMPLETE

The first status flag, LLAMA_STATUS_SUCCESS indicates that no failed ranks could be detected in *my_world_comm* so the application can proceed safely. LLAMA_STATUS_REPAIRED indicates that one or more ranks were found to have failed but they were successfully repaired, i.e., one or more ranks in *my_world_comm* has been replaced with one or more ranks from the pool of spare ranks and released from *CheckStatus()* together with all the other worker ranks. This status flag indicates that the user should take action to role back to a previous checkpoint idx at which a sufficiently resilient checkpoint was updated. LLAMA_STATUS_FAILED indicates that one or more ranks were found to have failed, but it wasn't possible to repair the communicator. This could happen if the llama::guard has run out of spare ranks or if the llama::guard detects a failure pattern for which none of its associated checkpoints may recover. This status flag will only be returned in the event that return_errors was set to true when creating the llama::guard, otherwise the application will write an error message on the std::cerr stream and exit with the EXIT_FAILED status. The final possible status flag LLAMA_STATUS_COMPLETE is only returned with a spare-rank and indicates that the application has completed and that the spare rank should proceed to clean-up.

In the event that a LLAMA_STATUS_REPAIRED status flag is returned, the user must specify how the application should be repaired. Llama simply exposes access to the content of all arrays that was marked as protected at the most recent checkpoint idx for which recovery is possible. To access the data content of an array at the most recent checkpoint idx, the method *RecoverArray()* is used.

Listing 11.10 – Method to recover data

```
my_llama.RecoverArray<class T>(const std::string &key, T* p_array,
  uint64_t dim1)
```

key is the key that was previously used to identify the array and *T* is the type of the objects stored in the array. *p_array* indicates to where the recovered data should be copied. For ranks that had not failed, this might very well be the same pointer, whereas for a newly introduced worker, rank relevant data structures must be created. As before *dim1* indicates the number of elements in the array. The same interface is used for 2D and 3D arrays

Listing 11.11 – Method to recover data

```
my_llama.RecoverArray<class T>(const std::string &key, T** p_array,
  uint64_t dim1, uint64_t dim2)
my_llama.RecoverArray<class T>(const std::string &key, T*** p_array,
  uint64_t dim1, uint64_t dim2, uint64_t dim3)
```

The example folder in the git repository contains several examples on how to make applications fault-tolerant.

Finalizing the Environment

Before finalizing the MPI environment, the Llama environment must be finalized using *my_llama.Finalize()* to clean up all checkpoints and to free all spares. If the environment is not finalized, unused data files may remain on the parallel file system.

Usage examples

A code example for the use of the Llama library is given in appendix E and additional examples are available in the Git repository. In the example in the appendix, a small code for finding a numerical approximation to the solution of the 2D Euler equation in parallel using MPI is made fault tolerant using the Llama library. Rank failures are forced by raising SIGKILL on a subset of nodes to demonstrate recovery in case of two different failure patterns. A detailed outline of the example is given in the code comments.

11.3 Numerical Experiments

Numerical experiments demonstrating the overhead initialization cost of creating the Llama Guard, adding checkpoints and marking arrays to protect are presented in sections 11.3.1, 11.3.2, and 11.3.2. In section 11.3.3, scaling tests on the walltime cost to periodically update checkpoints are presented. The cost of recovering data from a checkpoint is presented in section 11.3.4. All measurements are made for to-disk checkpoints, and for four different types of in-memory checksum checkpoints as outlined below

To-Disk Arrays to be protected are written to the GPFS filesystem using MPI-IO.

In-Memory G32P4 Arrays encoded in groups of 32 ranks. Four parity code blocks are created per code stripe. 14.3 percent memory overhead.

In-Memory G16P2 Arrays encoded in groups of 16 ranks. Two parity code blocks are created per code stripe. 14.3 percent memory overhead.

In-Memory G8P2 Arrays encoded in groups of eight ranks. Two parity code blocks are created per code stripe. 33.3 percent memory overhead.

In-Memory G8P1 Arrays encoded in groups of eight ranks. one parity code blocks are created per code stripe. 14.3 percent memory overhead.

The checkpoint types are listed in order of increasing resilience. The to-disk type checkpoints may recover lost data regardless of how many ranks have failed. In-memory checksum checkpoints may recover so long as the number of lost ranks within a group is smaller than the number of parity code blocks encoded and stored per stripe. Smaller groups with few parity code blocks may recover from fewer failure patterns and may thus be considered less resilient. Small groups with few code blocks are, however, expected to be faster to update as fewer operations and less communication between ranks in a group is needed to compute the checksum code blocks. Weak and strong scaling tests has been made from 112 to 7168 cores for all numerical experiments. The EPFL Fidis general purpose cluster was used to run the scaling tests. Each node in the cluster has two Intel Broadwell 2.6Ghz processors with 14 cores each, i.e. 28 cores per node. The cluster network fabric is Infiniband FDR fully-non-blocking, with a fat-tree topology.

11.3.1 Creating the Llama Guard

The first step in making an application fault tolerant by way of periodic check-pointing using Llama, is to declare a Guard object following the initialization of the MPI environment

Listing 11.12 – Declaration of the constructor for the Llama Guard

```
llama::guard::guard(MPI_Comm &input_comm, int num_spare_ranks, bool
  return_errors);
```

The first argument is a pointer to a duplicate of the world communicator, containing all worker and spare ranks. The second argument tells the Guard how many ranks to put aside in a pool of spare worker ranks. The third argument, if true, tells the Guard to throw error objects rather than exit upon encountering failure patterns which are not recoverable. The constructor of the Guard initiates various structures needed, and it removes *num_spare_ranks* ranks from the *input_comm* communicator and places these ranks in a pool for later use in case of failures. A scaling test is presented in Figurer 11.2. The walltime to create the Guard does not depend on checkpoints added later, and only increases weakly with the number of cores. The measured time varies substantially between measurements.

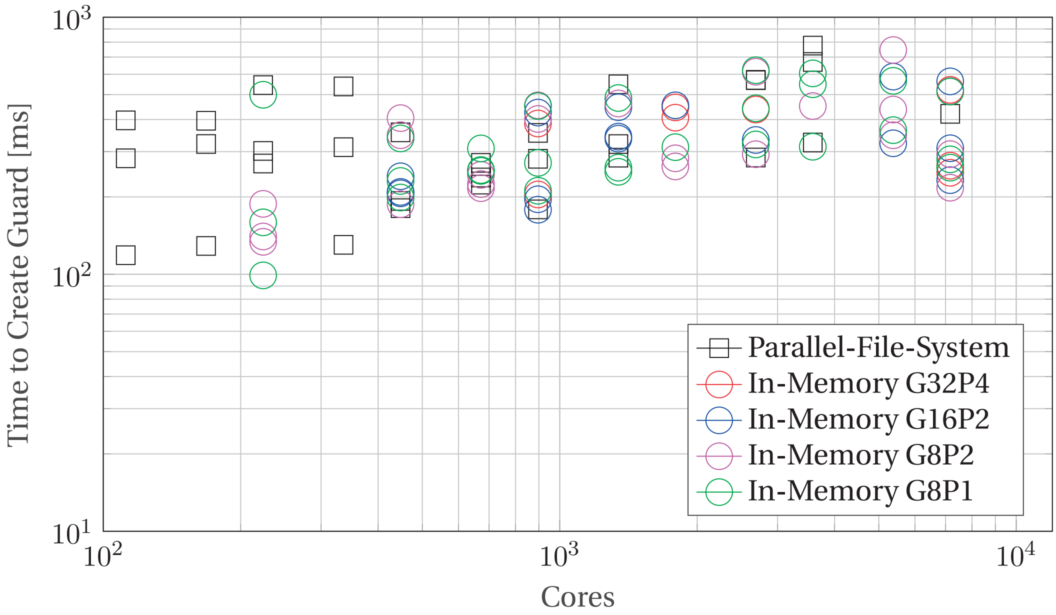


Figure 11.2 – Scaling test, time to create Llama Guard as a function of the number of cores as tested on the EPFL Fidis cluster. The time to create the checkpoint increases slightly with an increasing number of cores, but varies mostly due to difference in network load and job placement on the shared general purpose cluster.

11.3.2 Adding a Checkpoint to the Guard

For the Guard to protect arrays, one or more checkpoint objects must be associated with the Guard. To-disk and in-memory type checkpoints may be added using their respective constructors and passing a pointer to the Guard.

Listing 11.13 – Constructors for adding checkpoints

```
// Declaration of the constructor for the to-disk type checkpoint
void llama::checkpoint::disk::disk(llama::guard *MyGuard)
// Declaration of the constructor for the in-memory type checkpoint
void llama::checkpoint::memory::memory(llama::guard *MyGuard, bool
  copy_protected_data, int connected_ranks, int local_group_size, int
   num_parities_per_group)
```

Within the constructor for declaring a to-disk type checkpoint, only a few local initialization operations are needed. For the in-memory checksum type checkpoints, global operations are needed to verify that all workers are creating in-memory type checkpoint of the same group size, using the same number of parity blocks. In addition, an array of local communicators are created within each group. The overhead of attaching an in-memory checkpoint to the Guard is therefore higher than for the to-disk type checkpoint. As may be seen from the scaling test in Figure 11.3, the in-memory checksum checkpoint is an order of magnitude more expensive to initialize.

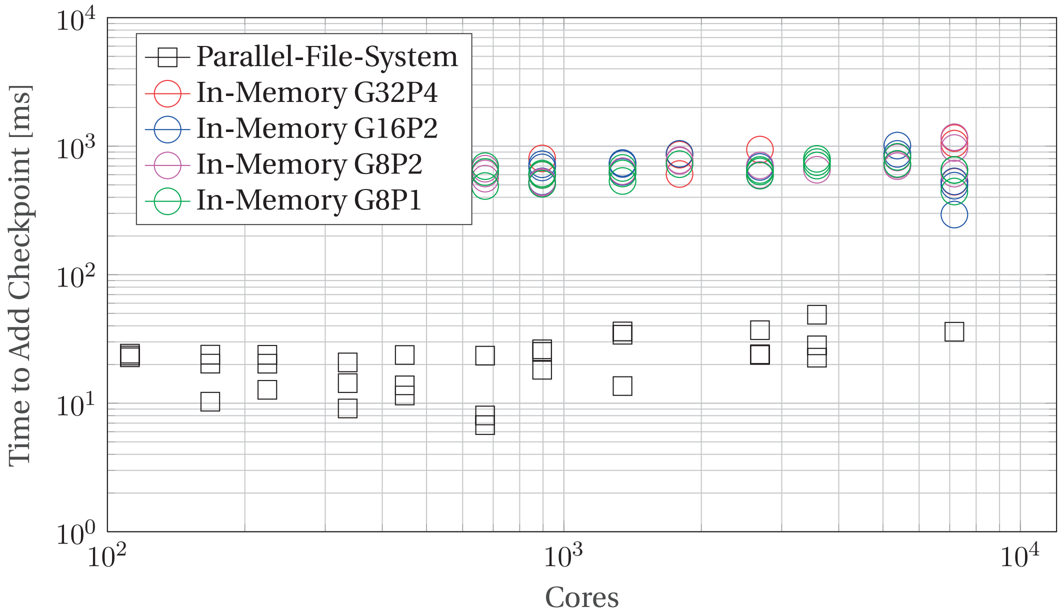


Figure 11.3 – Scaling test, time to attach a checkpoint-type to the Llama Guard as a function of number of cores. The time increases slightly with an increasing number of cores, the overhead of checking parameters and creating a matrix of local communicators makes the in-memory type checkpoint slower to initialize.

Adding an Array for the Guard to Protect

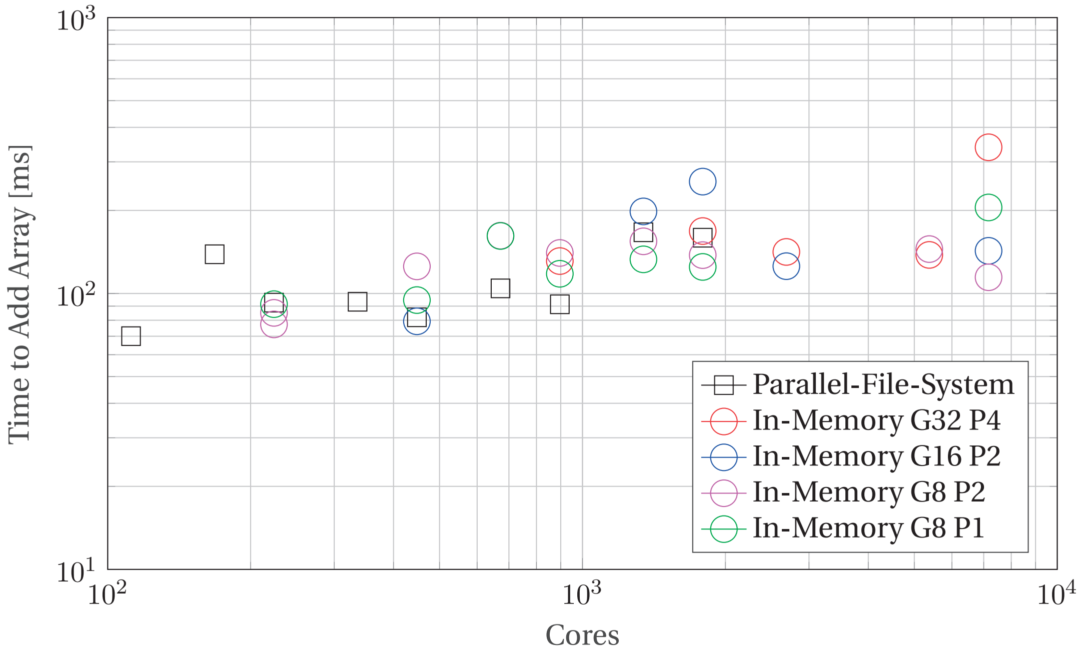
Any array, containing data needed upon recovery, must be protected by the Llama Guard created. To notify the Guard of the existence of the array, the *ProtectArray* method of the Llama Guard class is used. The method is a template on the data type of the array to be protected, and overloaded by the number of arguments to distinguish between 1D, 2D, and 3D arrays.

Listing 11.14 – Notifying the Llama Guard that an array must be protected

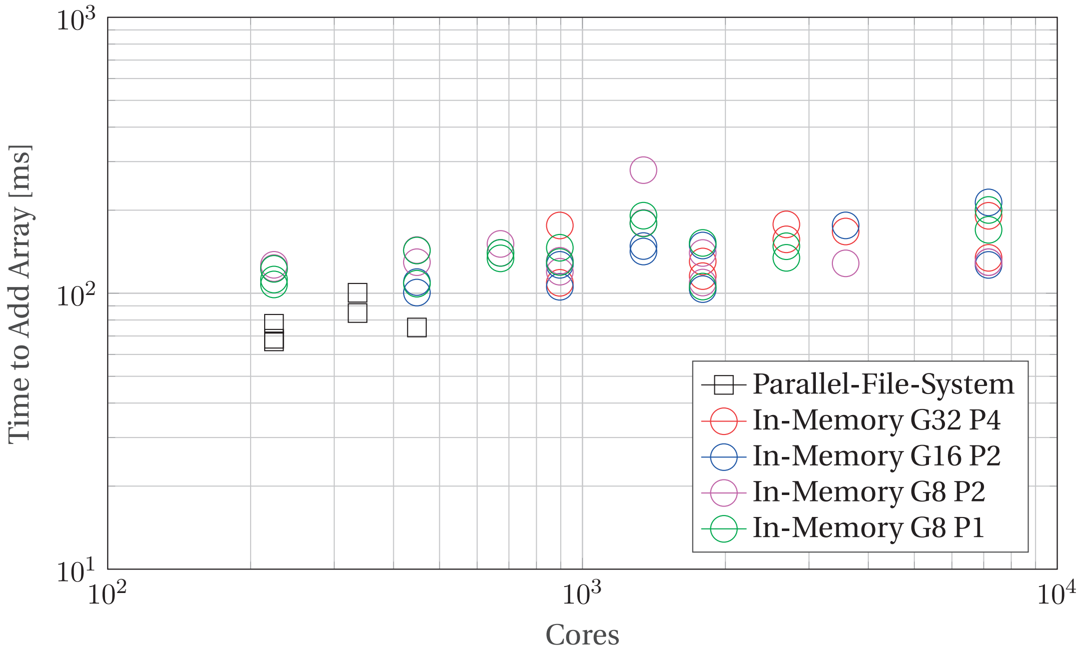
```
// Method for 1D arrays
template<class T> void guard::ProtectArray(const std::string &key, T
  *p_array, int dim1)
// Method for 2D arrays
template<class T> void guard::ProtectArray(const std::string &key, T
  **p_array, int dim1, int dim2)
// Method for 3D arrays
template<class T> void guard::ProtectArray(const std::string &key, T
  ***p_array, int dim1, int dim2, int dim3)
```

The method checks if the key supplied is unique- If it is not the application exists with an error code, or an error object is thrown, depending on how the Guard was initialized. If the key is unique, the array pointer and array dimensions are added together to a map connecting the key and the array information. Within the method, all checkpoints associated with the Guard is notified of the existence of a new array to be protected. Guard objects are friends of checkpoint objects, so checkpoints are notified by calling a protected method that initiates whatever structures that may be needed for the checkpoint type. Only two types of checkpoints exists, the to-disk that uses MPI-IO to protect an array on the parallel-file-system, and the in-memory which encodes checksum parity codes to protect the data and distribute these parity codes in a local group. In the case of the to-disk checkpoint, no action is needed. For the in-memory checkpoint, memory is allocated for the parity code blocks that will be written. If other types of checkpoints are to be implemented, they must inherit from the pure virtual *checkpointinterface* class and implement all required functionality.

In either case, before notifying checkpoints of the addition of an array to protect, a check is performed across all workers, verifying that all workers are adding an array with the same unique key. The operation is thus global and must be accessed by all workers to avoid deadlock. Figure 11.4 contains a weak and a strong scaling plot of the overhead walltime associated with adding an array for the Llama Guard to protect. The numerical experiment indicates that there is little or no difference in the cost of adding an array between the to-disk and the in-memory type checkpoints. The principal cost of the *ProtectArray* method is therefore likely the addition of a new array to the map, followed by the global agreement operation in the Guard that is independent of the size of the array.



(a) Weak Scaling – 12MB per Array per Core



(b) Strong Scaling – 128GB Distributed Array

Figure 11.4 – The wall-time cost of adding an array to the Llama Guard as measured on the EPFL Fidis general purpose cluster for 112 to 7168 cores. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times to illustrate the variance in timings due to job-placement and network load when using a shared cluster. The cost of adding an array is observed to be almost constant, only vaguely increasing with an increasing number of cores.

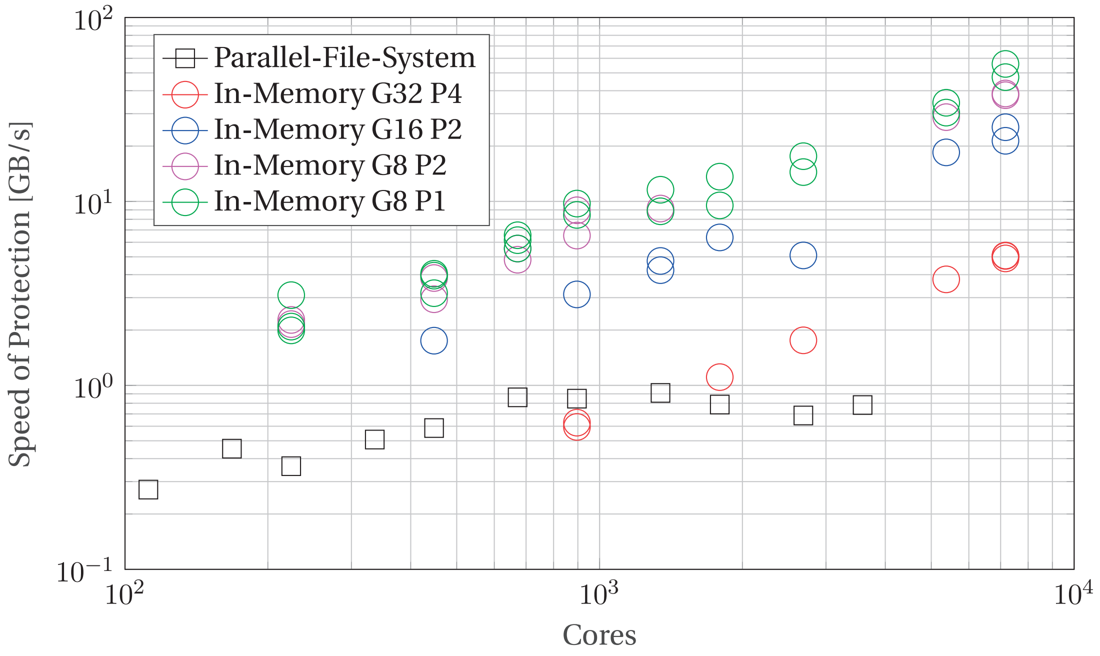
11.3.3 Cost of Updating Checkpoints

In the fault tolerant application, checkpoints protecting data will need to be periodically updated. This is done using the *Update* method. The method is a pure virtual function in the *checkpointinterface* class from which all checkpoints derive.

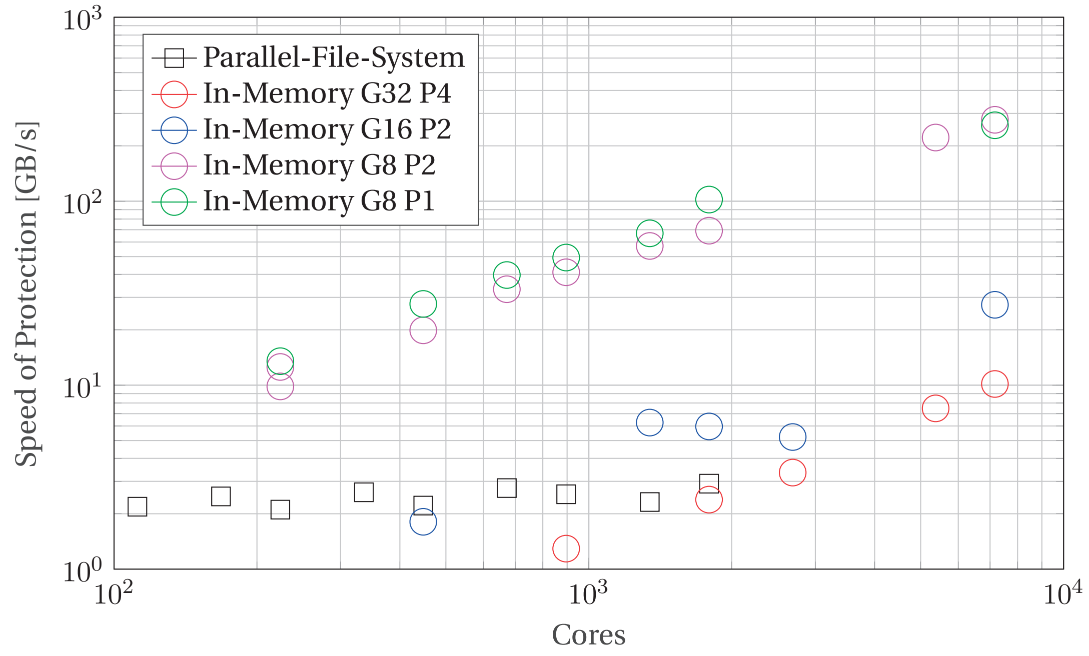
Listing 11.15 – Declaration for method to update checkpoint

```
// Update all data arrays and encode all checksums in checkpoint
void llama::checkpoint::memory::Update(int new_checkpoint_idx)
void llama::checkpoint::disk::Update(int new_checkpoint_idx)
```

When calling the function, the checkpoint will update the protection of all arrays marked by protection. The variable *new_checkpoint_idx* indicates the index for the new checkpoint. The index provided must be greater than any previous index used. The underlying action of *Update* depends on the checkpoint implementation. For the to-disk checkpoint, MPI-IO is used to write a copy of the array to be protected by the parallel file system. Once the array has been written, an agreement check is performed between all workers to verify that the new checkpoint is safe before deleting the file containing the previous checkpoint. For the in-memory checkpoint, all arrays are encoded using Reed-Solomon erasure code as outlined in section 11.2.1. The walltime cost of updating the checkpoint may be the most critical part of the Llama library as, unlike the creation of the Guard and adding arrays, the update will be performed periodically throughout the execution of the code to protect. Steps has therefore been taken to make the update as fast as possible. All essential checks on identifier keys and array dimensions has been moved to the *ProtectArray* method of the Guard class, used to notify the Guard of arrays to be protected. No checks are performed inside *Update*. The operation is global over all workers so deadlock will ensue if any workers are missing unless it being due to failure in which case it returns throwing an error object. Figure 11.5 contains weak scaling measurements on the EPFL Fidis cluster for 112 to 7168 cores for the protection of a 2D array. Measurements has been made for the to-disk type checkpoint as well as for four different types of in-memory checksum checkpoints. Each node on the Fidis cluster has 28 cores. In all four in-memory cases the number of adjacent nodes assumed likely to fail together is set to the size of a single node, 28, i.e., ranks $0, 28, 56, \dots$ will be placed in the zero'th group, rank $1, 29, 57, \dots$ in group 1 and so forth. At 12MB per core close to ideal scaling is observed when using a group size of eight with one or two parity codes per stripe. In the limit of using 7168 cores, Llama encodes data at a rate of 300GB/s. Which is more than two orders of magnitude faster than what may be achieved protecting the arrays using MPI-IO in the parallel-file-system checkpoint. Figure 11.6 contains strong scaling tests for a 16GB 2D array and a 128GB 2D array. In the strong-scaling test, the size of the 2D array per core decreases as the number of cores increases, making it more challenging to achieve perfect scaling. As in the weak scaling test, the in-memory checksum checkpoint is up to 200 times faster than the to-disk type checkpoint when using 7168 cores.

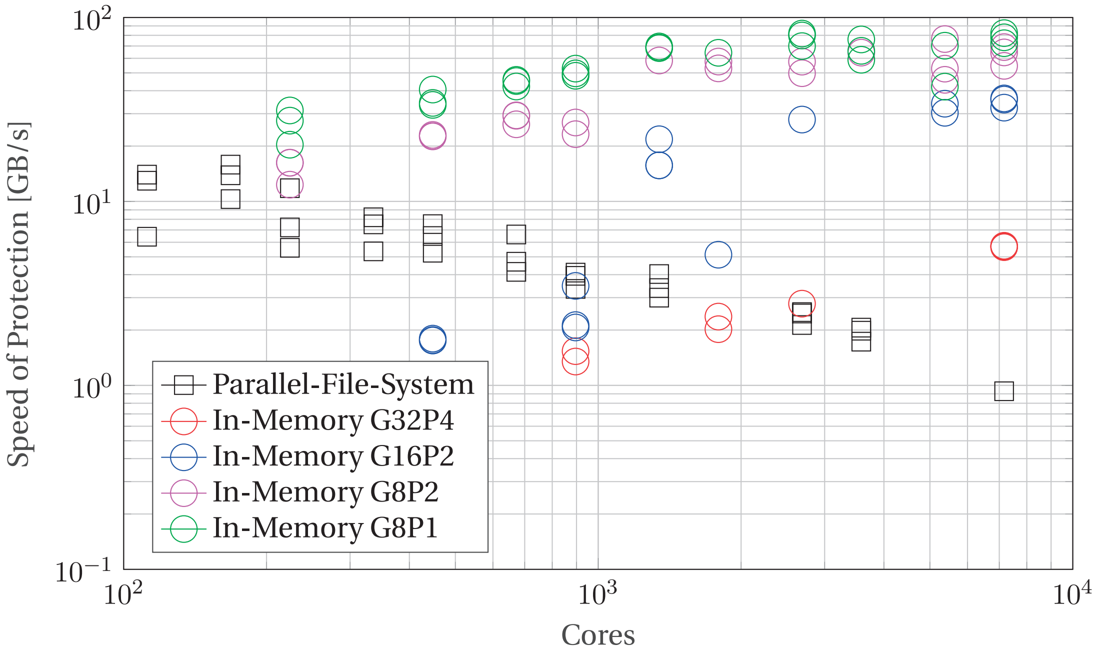


(a) Weak Scaling – 1.5MB per Array per Core

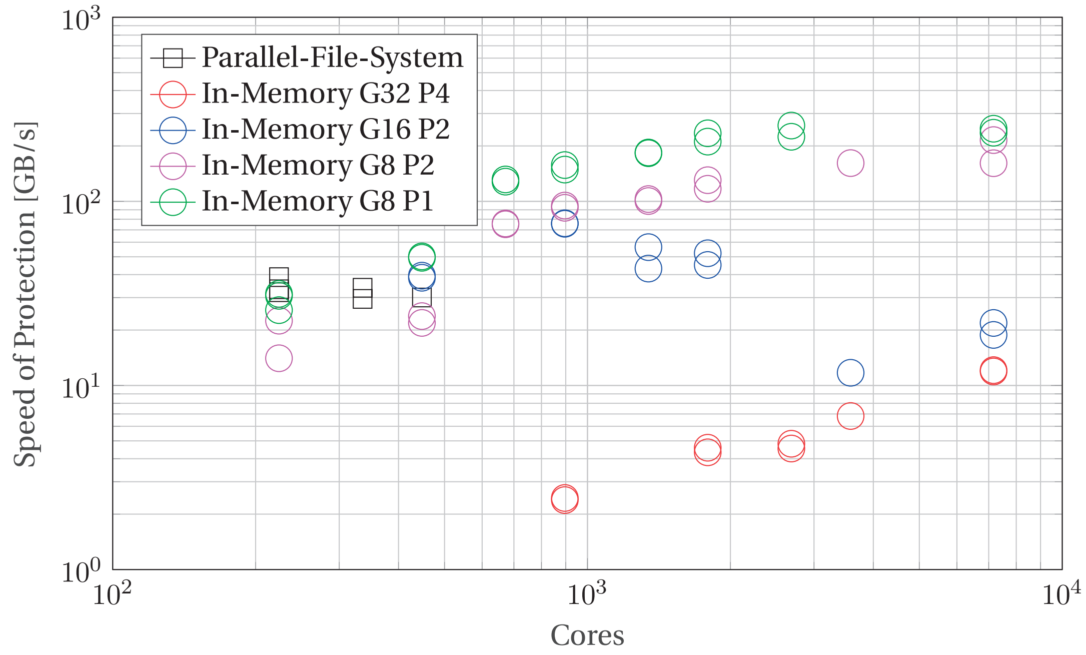


(b) Weak Scaling – 12MB per Array per Core

Figure 11.5 – Weak scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints.



(a) Strong Scaling – 16GB Distributed Array



(b) Strong Scaling – 128GB Distributed Array

Figure 11.6 – Strong scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding four parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system.

11.3.4 Cost of Data Recovery

If a status-check, using *CheckStatus* during the operation reveals that one or more ranks has been lost and replaced, the arrays at a previous *checkpoint_idx* may be recovered by the Llama Guard using the *RecoverArray* method of the Guard class. To identify the array to be recovered, one must use the key associated with the array, when the array was added to the Guard's list of arrays to protect using the method *ProtectArray* of the Guard class. The dimensions of the array, as well as a pointer *p_array* to where the newly recovered data from a previous checkpoint idx should be recovered, are part of the input arguments. This pointer can be a pointer to the same location as used when updating checkpoints associated with the Guard. The method is templated on the data type of the array to be recovered, and overloaded by the number of arguments to distinguish between 1D, 2D, and 3D arrays.

Listing 11.16 – Declaration for the method to recover array

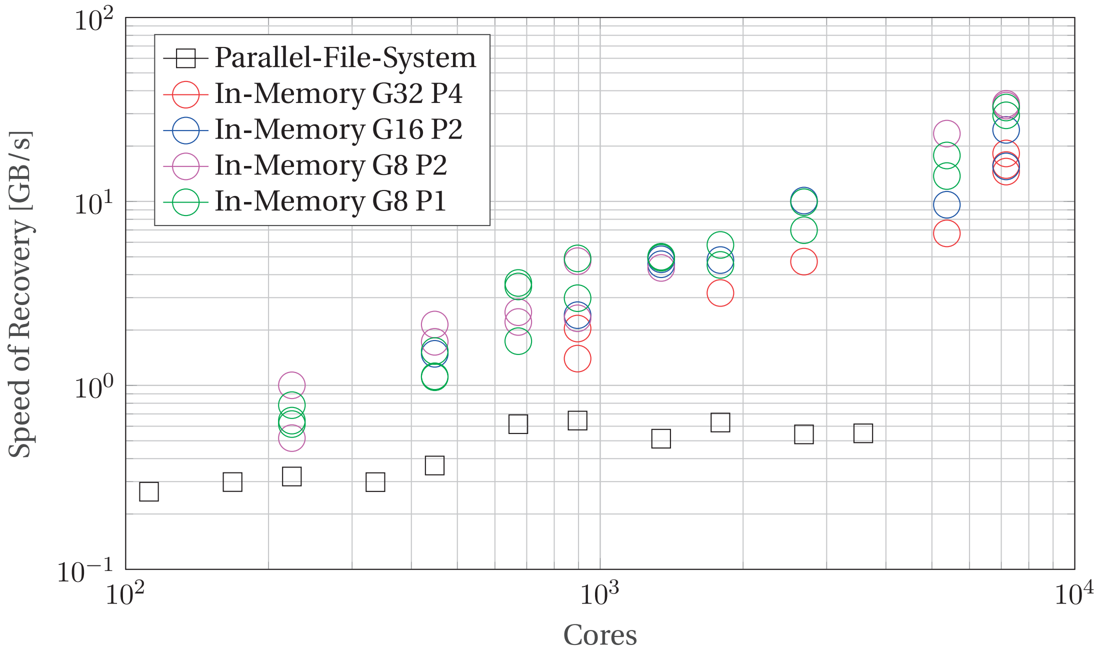
```
// Initiate recovery of 1D array
template<class T> void RecoverArray(int checkpoint_idx, const std::
  string &key, T *p_array, int dim1);
// Initiate recovery of 2D array
template<class T> void RecoverArray(int checkpoint_idx, const std::
  string &key, T **p_array, int dim1, int dim2);
// Initiate recovery of 3D array
template<class T> void RecoverArray(int checkpoint_idx, const std::
  string &key, T ***p_array, int dim1, int dim2, int dim3);
void llama::checkpoint::disk::Update(int new_checkpoint_idx)
```

When a Guard is asked to recover the content of an array at some previous index, it first performs an agreement function, ensuring that all workers have been asked to recover the same array. This is followed by a few basic checks that the array exists and that the information on key, array data type, and dimensions is consistent with an array being protected. The Guard then cycles over all checkpoints type associated with it, and queries the checkpoint to see if it is able to recover all data at the checkpoint index specified. Among the candidates, it choses the checkpoint which it reckons is the fastest to recover from. The chosen checkpoint is then instructed by the Guard to recover the data to the location of the pointer supplied. How recovery is performed depends on the checkpoint type. To-disk checkpoints will use MPI-IO to read from the parallel-file-system, while in-memory checksum checkpoints will perform a group-local decoding procedure.

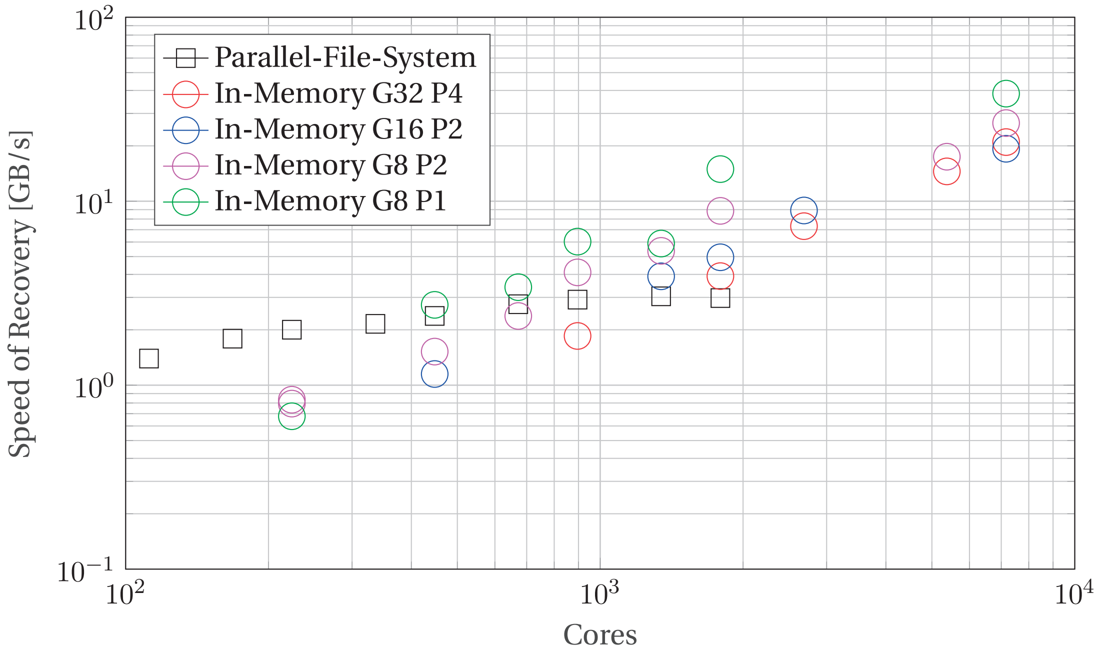
Figure 11.7 contains weak scaling numerical experiment, demonstrating the rate at which a 2D array may be recovered using the different types of checkpoints. Tested for 112 cores to 7168 cores on the EPFL Fidis cluster. Two scaling tests are demonstrated, one with 1.5MB per core and another with 12MB per core. Each data point constitutes a job on the shared general purpose cluster. Each experiment has been performed three times since job placement and cluster network load may substantially impact the speed of recovery. Figure 11.8 contains strong scaling measurements using a fixed global array size of 16GB in Figure 11.8a and 128GB in Figure 11.8b. In all tests, a single rank in a single group is terminated by raising a SIGKILL locally on the rank to disappear. For recovery from in-memory checkpoints, all groups containing a broken rank must decode in parallel across the group to recover the lost data. All unbroken groups may simply copy the unbroken content of the local array. The time-to-recover is therefore limited by the time it takes the broken groups to decode all local data blocks. The location of ranks within a group on the cluster and the network may therefore substantially impact the speed of recovery. For to-disk checkpoints, all cores must read the array from a file on the parallel-file-system, regardless of whether or not they were involved in a failure or not. This makes the to-disk checkpoint time-to-recovery less sensitive to how ranks are distributed on the cluster. Instead the speed of recovery is sensitive to the load on the parallel-file-system at the time of the job execution.

The recovery procedure always proceeds after a status check where one or more worker ranks are found to have failed. The status check is a method of the Llama Guard class. When worker ranks enter the *CheckStatus* method, an agreement operation is performed across all workers to check if any workers have failed. If one or more worker ranks are found to have failed, spare ranks are released and the worker communicator revoked and repaired. During the repair, spare ranks are injected to take the place of lost worker ranks. The process of detecting failures and repairing the worker communicator is essential for the automatic restart at an earlier checkpoint index. The method is implemented using functionality of ULFM-MPI. Unfortunately we've found that when testing the Llama library using a large number of cores on the EPFL Fidis cluster, the detection of failed ranks and subsequent communicator repair sometimes experience either a deadlocks or fail completely. Following the simulation of rank failures by raising SIGKILL on a rank to terminate it, deadlock or failure of otherwise failure-free ranks often happen within *MPIX_Agree* or *MPIX_Comm_Replace*. The errors returned vary in type. Sometimes failure-unaffiliated ranks lose contact with the MPI helper daemon orted and exit, other times errors related to the operations on the InfiniBand network are returned. As described in section 11.1.3, ULFM-MPI is still at an early phase of development and is not part of the official MPI standard yet. According to the developers of ULFM-MPI, the possibility of buggy corner cases is high.

In our current implementation using ULFM-MPI 2.0, correct detection of failures and subsequent repair of the worker communicator happen only about half the time when running large jobs with 1000+ cores. When the detection and repair proceeds correctly, the time to repair ranges from less than one second to 80+ seconds. Due to the inconsistent behavior of the detection and communicator-repair operations, we're unable at the moment to perform meaningful large-scale scaling tests on the cost of this component of Llama.

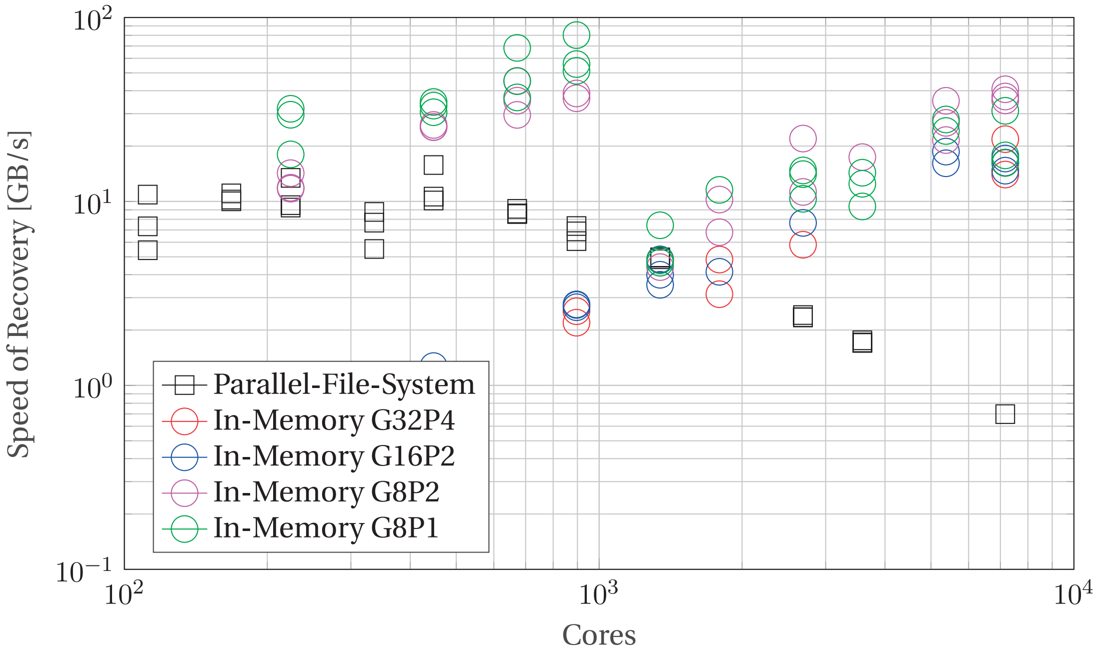


(a) Weak Scaling – 1.5MB per Array per Core

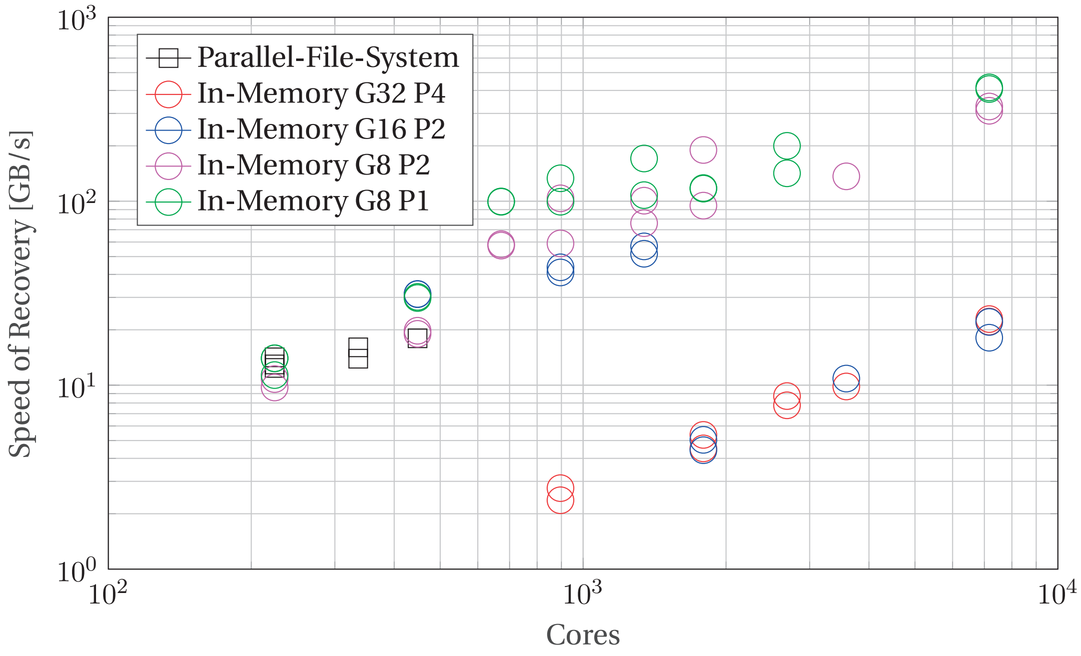


(b) Weak Scaling – 12MB per Array per Core

Figure 11.7 – Weak scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints.



(a) Strong Scaling – 16GB Distributed Array



(b) Strong Scaling – 128GB Distributed Array

Figure 11.8 – Strong scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding 4 parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system.

11.4 Summary

The Llama library delivers fault tolerance in HPC trough checkpointing. It is the first library to support both automatic rollback without restart and the use of an arbitrary number of checkpoint levels with topology aware checksum checkpoints of arbitrary group size and number of parity code blocks. In section 11.3.3, it was demonstrated that the library can achieve in excess of 300GB/s encoding when running on 256 nodes, similar to what has been demonstrated using the SCR library by LLNL[217], with the major distinction that there are no limitation in Llama on the group size and the number of parities to use per group.

One issue not touched upon in outlining the recovery procedure is that for any application, newly joined worker ranks must create all structures that was created by the original worker ranks during initialization. This initialization phase may involve communication with other worker ranks and could potentially be very time consuming. A potential path around this issue was presented in [49] where it was demonstrated that this procedure need not be slow nor cumbersome to implement. By logging all MPI transactions during the initialization phase and protecting these, new workers can recover in complete isolation from other workers waiting to restart at the previous checkpoint.

The library is still under development and there are several points that needs to be addressed in future versions. At the moment, only the use of spare nodes to replace failed workers has been implemented, i.e., there is no support for spawning new ranks as needed. Another issue that remains is that upon injecting a large number of errors, the failure-detection and communicator repair may fail. Identifying the underlying cause of these issues is ongoing work.

Finally, the implementation of error handling within Llama functionality is not complete, so at the moment if worker ranks fail within Llama calls such as the update or recovery methods of the Guard, this will result in undefined behavior. To protect against failures during the process of updating in-memory checkpoints, the approach presented in [295] is to be used, maintaining an extra copy of the parity code blocks during checkpoint update.

12 Partial Information Recovery with Incomplete Checksums

Hardware is inherently failure prone. For data centers and cloud services, protecting user data has been essential for as long as the services have existed. In early days, the standard approach was protection trough triple redundancy. At any given moment, three separate hard drives on separate systems would store any given data[202, 166]. If at some point, a drive would fail, the two remaining hard-drives are to make a new copy by sending their data over the network to a new failure-free machine. The process of copying the full content of a hard-drive over network fabric might take an hour or so, depending on the state of network congestion.

With triple redundancy, three hard-drives on separate systems would have to all fail within the span of roughly an hour for the data to be lost. Consumer grade hard disk drives (HDD) have an annualized failure rate of 2 to 5 percent when running 24 hours a day, 365 days a year[270]. With these ballpark numbers, the probability of loosing one hard-drive worth of data within a year is in the order of $\sim 10^{-15}$. Thus, even with a very large number of hard-drives, the probably of loosing data becomes small. Newer Solid State Drives (SSD) are unlikely to lessen the need for failure protection techniques meaningfully. Studies have shown that they have lower annual replacement rate of 1-3 percent, but in turn they have a higher rate of uncorrectable data errors [271]. The problem with the approach of redundancy for protecting data is that it is very expensive. The main cost of running a data-center is due to energy consumption and hardware purchase. Triple redundancy essentially triples the cost of running the service.

For said reasons, the industry no longer rely solely on redundancy as a mean of fault-tolerance. Instead, erasure codes have found use. Erasure codes are a form of forward error correction that take data consisting of k symbols and turn it into a larger data set with n symbols such that the original data may be recovered from a subset of the n symbols. Erasure codes for which any k symbols are sufficient to recover the original data are called optimal, these codes are as resilient as possible, but typically scale quadratic in terms of coding and decoding complexity with respect to n. The $n-k$ extra symbols created are typically referred to as parity codes or checksums. The most commonly used erasure code is Reed–Solomon error correction[249], used in RAID6 and in various storage services.

The advantage of using Reed-Solomon erasure code instead of redundancy is that it is relatively memory efficient. Imagine three hard-drives full of data that one would like to protect, i.e. $k = 3$, say another three hard-drives are used to store parity code, i.e. $n = 6$. We'd have to loose four drives among the six, before the data on the lost hard-drives become truly lost. This comes at the cost of a 100 percent overheard in hard-drive usage. Compare this to the triple-redundancy approach, here nine hard-drives would be required corresponding to a 200 percent overheard, this despite the two approaches having essentially the same probability of data-loss. The Reed-Solomon approach is thus very efficient in terms of disk-usage. The trade-off is that Reed-Solmon requires encoding and decoding of data, which increase both CPU usage and network congestion. In practice, major data-centers are typically using various forms of hierarchies of parity codes, in combination with redundancy techniques, to mitigate issues of network congestion in particular[165, 267, 238].

The use of erasure code is beginning to find its way into HPC as well. In the context of fault-tolerance, the interest is to protect data in-memory to avoid the comparatively slow parallel-file-system; this makes the memory conserving property very attractive. In addition, most clusters used for large-scale applications are equipped with high-bandwidth low-latency networks, which serves to somewhat alleviate the cost of encoding data. In Chapter 11, erasure codes were used for this exact purpose to enable a library for fault-tolerance to create memory-conserving in-memory checkpoints.

Unfortunately Reed-Solomon too has limits, even optimal erasure codes can not recreate lost data if the number of lost hard-drives, or nodes, k_l is greater than $n-k$. Up until this point, all data lost can be recreated with bit-wise accuracy, but after, there's no known way of recreating the lost data. This is unfortunate. In the context of HPC fault-tolerance trough light-weight checkpointing, as demonstrated in chapter 11, this means one must revert to a more resilient checkpoint such as one written to the parallel-file-system. In the context of data-centers and cloud services protecting costumer data, it is unfortunate because they are forced to use sufficiently many resources to ensure that the probability of this situation happening is infinitely small.

The decoding procedure can be thought of, in very simple terms, as the procedure of solving a linear system. In order to recover lost data, one must solve a system for which the number of columns is the number of lost symbols k_l, while the number of rows is the number of parity code symbols $n-k$. The reason that it becomes impossible to recover the data when $n-k < k_l$ is that the system to solve become under-determined. If real numbers were used in the encoding scheme, infinitely many solutions to the decoding procedure would exist. If, as in the case of Reed-Solomon, the encoding matrix and symbols to encode str in a Galois field, the number of possible solutions would still be finite, but there would be no unique solution.

Despite the fact that no unique solution exists to the decoding problem when $n-k < k_l$, one could argue that the remaining equations still hold information about the structure of the lost data. Even though the number of possible solutions is infinite, the number of solutions not admissible is likewise infinite. One might speculate that given some other knowledge about the data that was decoded, say knowledge of how one element in a vector tends to relate to another, could this information be used to impose other conditions in such a way that the system to solve yet again become well posed? In this chapter we present a small preliminary study demonstrating a proof-of-concept. For reasons of simplicity, we consider an erasure code in the style of Reed-Solomon, but on real numbers.

12.1 The Weighted Checksum Scheme

Since real numbers can only be represented with finite precision on computers, most erasure codes such as Reed-Solomon, are designed to be applied to data represented as elements in a Galois field so that bitwise exact recovery is possible, thereby allowing the encoding/decoding mechanism to be agnostic with respect to what the bits represent.

In HPC fault-tolerance applications, some form of numerical data is often what needs to be protected. A vector, or matrix, filled with real numbers. This data could be treated as bit-streams using Reed-Solomon encoding, but parity codes could also be generated directly from the floating point numbers. In [189] the authors list a number of advantages in doing so. A main argument is that one avoids the trouble of introducing Galois Field arithmetic in the encoding and decoding procedure, instead being able to rely on standard matrix operations for floating point numbers. The disadvantage is the introduction of round-off errors during the recovery procedure due to the limited precision with which floating point numbers may be represented. The latter may however be of limited concern since it has been shown that the loss of accuracy can be limited with a clever choice of checksum encoding matrix [64, 189].

In the introduction below, we use k to indicate the number of separately stored data, m to indicate the number of parity codes to compute, i.e. checksums, and $n = m + k$ the total number of data and code blocks. In the context of the example given in the previous section, this corresponds to a total of n hard-drives, among which k separate hard-drives are used to store the data that must be protected, and m hard-drives are used to store checksum parity code.

Consider a vector $\bar{x}$ of length d times k. Let's assume that this vector is partially stored in k equal parts on k independent hard-drives. Then the "sub-vector" stored on each hard-drive contain d elements. For ease of notation, let $\bar{x}^i$ denote the i'th sub-vector. A simple way of protecting the content of the vector against hardware failures is to compute an element-wise sum, i.e. a new vector $\bar{c}$, also containing d elements,

$$\bar{c} = \sum_{i=1}^{k} \bar{x}^i \tag{12.1}$$

and then storing $\bar{c}$ on a separate hard-drive, $m = 1$. We will refer to the vector $\bar{c}$ as the checksum hence forward. If at some point we were to lose a hard-drive $k_{lost} \in \{1, k\}$. No matter which one it is, the vector $\bar{c}$ can be used to recover the data trough a summation on the form

$$\bar{x}^{k_{lost}} = \bar{c} - \sum_{i \neq k_{lost}}^{k} \bar{x}^i \tag{12.2}$$

The above approach is extendable to create a simple scheme for protection against multiple failures, called a weighted checksum scheme. Suppose that we can afford to store m checksum vectors on m separate hard-drives, we would then compute each checksum vector $\bar{c}^i$ as such

$$\begin{cases} a_{1,1}\bar{x}^1 + \ldots + a_{1,k}\bar{x}^k & = \bar{c}^1 \\ & \vdots \\ a_{m,1}\bar{x}^1 + \ldots + a_{m,k}\bar{x}^k & = \bar{c}^m \end{cases} \tag{12.3}$$

where $a_{m,k}$ are some weights to be chosen. The matrix $A = (a_{i,j})_{m,k}$ is called the checksum matrix. If multiple hard-drives fail, how could the data be recovered? Assume, without loss of generality, that all of the first k_{lost} hard-drives have failed, and the all subsequent hard-drives have survived, we may then derive an equation to recover the sub-vectors $\bar{x}^1, \bar{x}^2 \ldots, \bar{x}^{k_{lost}}$ lost by restructuring (12.3) to arrive at

$$\begin{cases} a_{1,1}\bar{x}^1 + \ldots + a_{1,k_{lost}}\bar{x}^{k_{lost}} & = \bar{c}^1 - \sum_{t=k_{lost}+1}^{k} a_{1,t}\bar{x}^t \\ & \vdots \\ a_{m,1}\bar{x}^1 + \ldots + a_{m,k_{lost}}\bar{x}^{k_{lost}} & = \bar{c}^m - \sum_{t=k_{lost}+1}^{k} a_{m,t}\bar{x}^t \end{cases} \tag{12.4}$$

We refer to the coefficient matrix of the left hand side matrix in the above linear system, consisting of k_{lost} columns of A, as A_r. For recovery to always be possible, the coefficients of the weighted checksum matrix A must be chosen in such a way that for any possible A_r, a unique solution is guaranteed to exist. I.e., the elements in the checkpoint matrix A must be chosen so that any sub-matrix of A is non-singular as this guarantees that A_r will always have full rank. Many structured matrices such as Vandermonde matrix, Cauchy matrix, and Gaussian Random matrix satisfy this condition. Not all such matrices would necessarily be suited though, it is also important that any sub-matrix A_r is well conditioned. If A_r has a high condition number, round off errors will accumulate and reduce the accuracy of the recovered data[18]. Gaussian Random matrices are both well conditioned and satisfy the condition that any submatrix is non-singular[96]. They are therefore a natural choice as noted in [189], and will be used as the checksum matrix for all numerical experiments presented in this chapter.

12.2 Partial Information Recovery for Incomplete Checksums

The unfortunate limitation of the approach, as with all erasure codes, is that if the number of hard-drives lost k_{lost} is larger than the number of checksums m, there is no unique solution to the problem of recovering the lost sub-vectors $\bar{x}^1, \bar{x}^2 \ldots, \bar{x}^{k_{lost}}$ since the system (12.4) becomes under-determined. Of course, one could simply let m be very large to avoid that situation, but increasing m means increasing the overhead in terms of hardware and energy. Ideally, we'd like to keep the overhead due to data protection as low as possible. Therefore, it would be immensely practical if we could somehow magically find the right $\bar{x}^1, \bar{x}^2 \ldots, \bar{x}^{k_{lost}}$ among the infinite number of solutions to the under-determined system (12.4) when $m < k_{lost}$.

To appreciate how that might be possible, let's take a step back and consider again the case of having only a single checksum vector. If we are so unfortunate to loose $k_{lost} > 1$ number of hard-drives, the solution space of every element d in each lost sub-vector $\bar{x}^1, \bar{x}^2 \ldots, \bar{x}^{k_{lost}}$ is spanned by an $k_{lost} - 1$ dimensional affine hyperplane as can be deduced from (12.3). The solution space is infinitely large, so we can not naively recover the lost sub-vectors by direct computations. Here's an idea though, let's assume that the data vector $\bar{x}$ stored in a distributed manor on many hard-drives has some structure to it, and that we have some knowledge of what that structure is. Say, we might be informed that the content of the vector $\bar{x}$ represents points sampled from a $\mathcal{C}^\infty$ functional. Now, given this knowledge, what if, among the infinitely many solutions to (12.3), for each element in each of the lost data vectors $\bar{x}^1, \bar{x}^2 \ldots, \bar{x}^{k_{lost}}$, we choose the solution which makes, in some yet to be defined sense, the function that $\bar{x}$ represents as smooth as possible?

The essence of the approach is simple. Let $\dot{\bar{x}}$ represents the first order derivative of $\bar{x}$; now, find the sub-vectors $\bar{x}^1, \bar{x}^2 \ldots, \bar{x}^{k_{lost}}$ that minimize $\|\dot{\bar{x}}\|_2$ under the constraint that all checksum equations must be satisfied. By writing $\dot{\bar{x}}$ as a function of $\bar{x}$ with the application of a finite difference stencil, and then rewriting the checksum equations (12.3) to depend on $\dot{\bar{x}}$ instead of $\bar{x}$, we are left with a well-posed convex optimization problem for $\dot{\bar{x}}$ that may be solved using standard methods. The approach worked fairly well, though the derivation is somewhat long, and becomes especially involved if to be extended to surfaces or volumes. In addition, due to the global nature of the optimization, it is computationally expensive. Upon further experiments, we found that though the underlying idea was right, a slightly different path proved better.

Instead of formulating a problem that finds the smoothest possible solution that satisfy all checksum equations, we assume that some compressed data or reduced model of $\bar{x}$ is available, denoted $\tilde{x}$. We then formulate a different optimization problem, i.e., find the sub-vectors $\bar{x}^1, \bar{x}^2 \ldots, \bar{x}^{kk_{lost}}$ that minimize $\|\bar{x} - \tilde{x}\|_2$ under the constraint that all checksum equations must be satisfied. This approach turns out to both work well and, unlike the other approach, be particularly simple to formulate and solve. The method is introduced in section 12.2.1, and extended into an iterative scheme in section 12.2.2 to demonstrate partial data recovery for incomplete checksums of image data.

12.2.1 Minimizing Distance to Inexact Data

Let's define $\hat{c}$ as the right hand side of (12.4), so that the equation for recovery may be written as

$$A_r \bar{x}^{1:k_{lost}} = \hat{c} \tag{12.5}$$

When $m < k_{lost}$, all possible admissible solutions z to the under-determined system above may be written as

$$\bar{z} = A_r^+ \hat{c} + \left[I - A_r^+ A_r \right] \bar{w} \tag{12.6}$$

where A_r^+ is the Moore-Penrose pseudo inverse of A_r. $\bar{w}$ is an arbitrary vector with k_{lost} elements. The solution corresponding to $w = \bar{0}$, is the minimum norm solution, i.e., the solution for which $\|z\|_2$ is the smallest amongst all the admissible solutions. If $\bar{x}$ approximates a function that is never too far from zero, one might speculate that computing

$$\bar{x}^{1:k_{lost}} = A_r^+ \hat{c} \tag{12.7}$$

could potentially recover data that is close to what was lost. Let's take that approach a bit further, imagine that we have access to some reduced model or compressed data $\tilde{x}$ that approximates the original vector $\bar{x}$. Upon failure, we'd like to use this data in conjunction with the checksum equations to find an even better approximation to the lost data. Subtracting $A_r \tilde{x}^{1:k_{lost}}$ from (12.5) we recover

$$A_r \left(\bar{x}^{1:k_{lost}} - \tilde{x}^{1:k_{lost}} \right) = \hat{c} - A_r \tilde{x}^{1:k_{lost}} \tag{12.8}$$

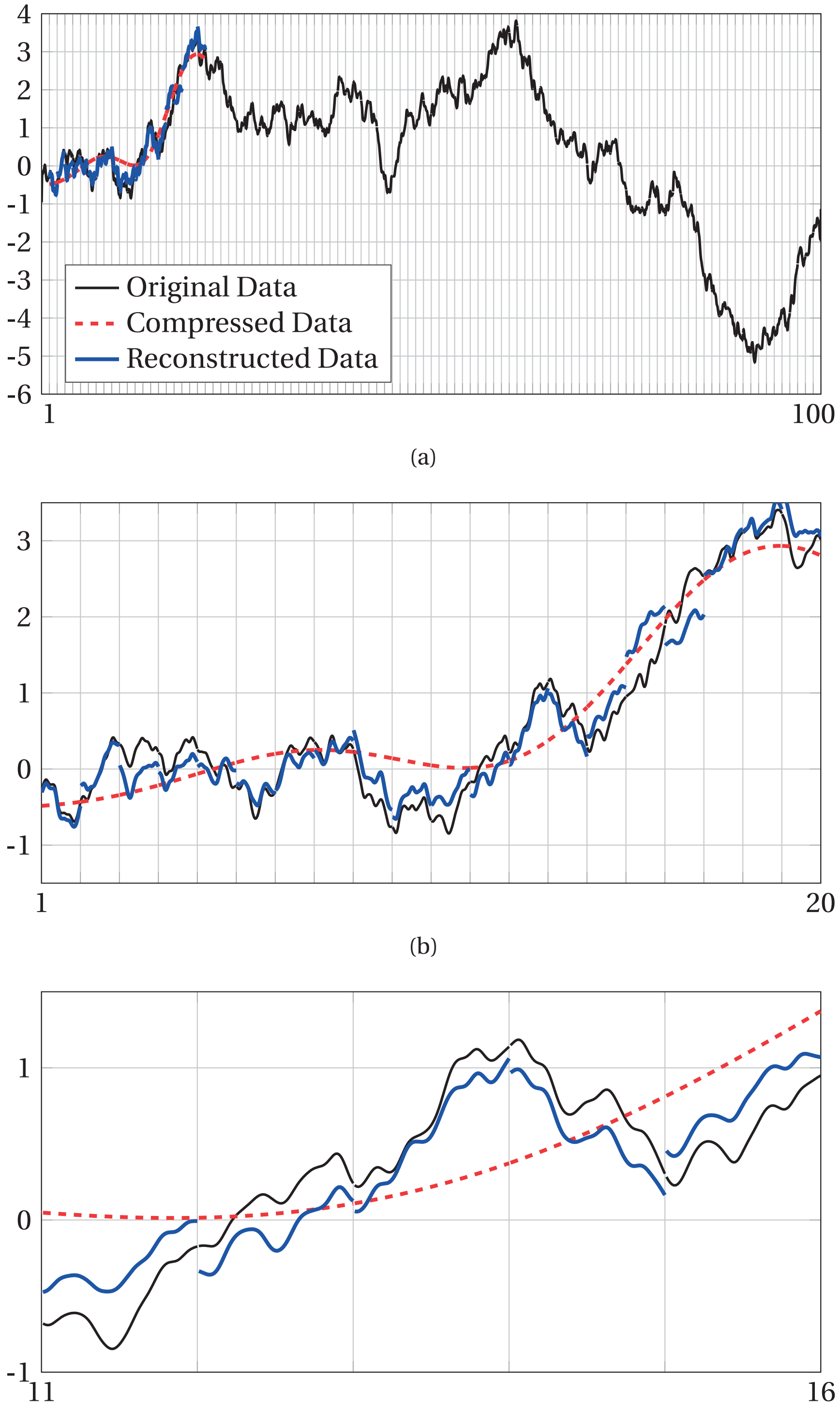
Applying the M–P pseudo inverse A_r^+ on each side and isolating $\bar{x}^{1:k_{lost}}$ we arrive at

$$\bar{x}^{1:k_{lost}} = A_r^+ \hat{c} + \left[I - A_r^+ A_r \right] \tilde{x}^{1:k_{lost}} \tag{12.9}$$

where $\bar{x}^{1:k_{lost}}$ as computed in (12.9) is the solution that minimize $\left\|\bar{x}^{1:k_{lost}} - \tilde{x}^{1:k_{lost}}\right\|_2$.

Two experiment with this approach at partial information recovery for incomplete checksum equations are presented in Figures 12.1 and 12.2. The vector $\bar{x}$ contains data created from a smoothed random walk, it contains a total of 10.000 data points. The data vector $\bar{x}$ was split into $k = 100$ separate sub-vectors, each containing 100 data points, and protected trough the creation checksum vectors as in (12.3), encoding vectors element-wise. $\tilde{x}$ is created by taking the first 20 modes of a Fourier transform of the entire dataset $\bar{x}$. After encoding the m checksum vectors, the first $k_{lost} = 20$ sub-vectors were removed. The figure contains both the original data vector $\bar{x}$, the compressed data $\tilde{x}$, and the recovered data $\bar{x}^{1:k_{lost}}$ computed by (12.9). In Figure 12.1, the data was protected using $m = 15$ checksums, hence in the incomplete data recovery, 33.3% more sub-vectors were lost than checksum vectors created. In Figure 12.2, the data was protected using $m = 18$ checksums, hence in the incomplete data recovery, 10% more sub-vectors were lost than checksum vectors created. For all experiments, the checksum matrix A was a Gaussian Random matrix.

This preliminary result indicates that the method works well. An interesting aspect of the approach of ensuring uniqueness by minimizing the distance between the data to be recovered, and some inexact data, is that the algorithm may be used to feed itself in an iterative manor. An iteration could consists of first solving the constrained minimization problem, followed by the application of some regularization function that modifies the data towards some property, smoothness for example. In the next section, the method just derived is used to create an improved, iterative, self-feeding algorithm for incomplete data recovery.

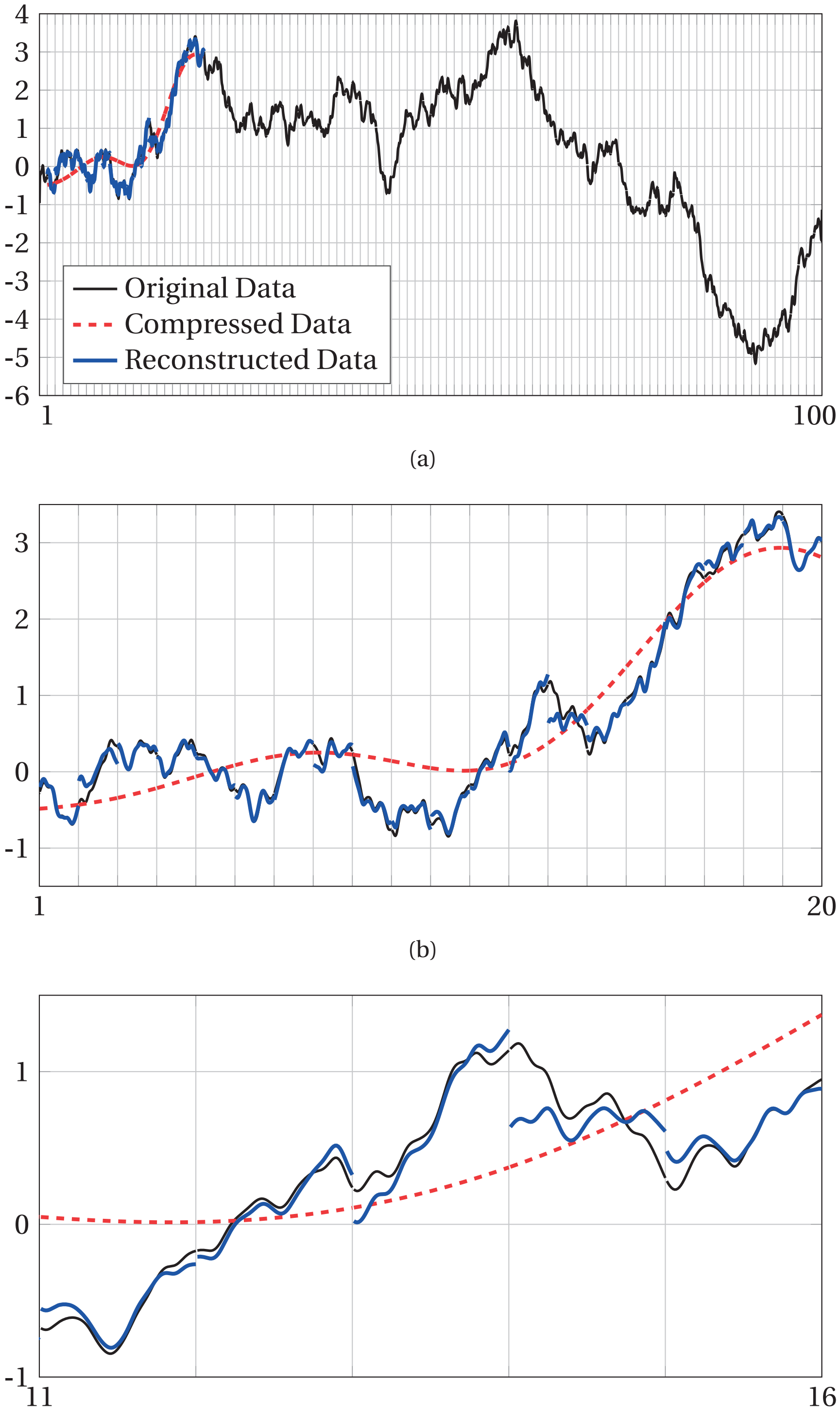


(a)

(b)

(c)

Figure 12.1 – Numerical experiment testing the method (12.9) for partial information recovery in incomplete checksums. The data indicated by the black line was stored in $k = 100$ separate containers. $m = 15$ checksum vectors were created to protect the content of the containers. The $k_{lost} = 20$ first containers are removed, i.e. 33% more data vectors are lost than checksum code vectors created.



(a)

(b)

(c)

Figure 12.2 – Numerical experiment testing the method (12.9) for partial information recovery in incomplete checksums. The data indicated by the black line was stored in $k = 100$ separate containers. $m = 18$ checksum vectors were created to protect the content of the containers. The $k_{lost} = 20$ first containers are removed, i.e. 11% more data vectors are lost than checksum code vectors created.

12.2.2 An Iterative Recovery Scheme

In the previous section we demonstrated that partial information recovery for incomplete checksums is possible when some compressed version, or reduced model, of the lost data was available to be used in conjunction with the under-determined checksum equations.

Here we present a further improved method in the form of an iterative algorithm. Computing (12.9) reconstructs the lost data by finding the solution, nearest to some guess, that satisfy the checksum equations. After performing this operation, one could continue the recovery procedure by applying some function, to the approximation found, that filters or modifies the solution in accordance with what meta knowledge one might have on the structure of the lost data. If, for example, it is known that the data to be recovered represents a continuous surface, one could apply a function that smooth the solution a bit.

This new, filtered solution, will however in all likelihood no longer satisfy the checksum equations. So it could be fed back into (12.9) as a new, improved guess. In this way, one could alternate between the two, to potentially arrive at a better approximation to the data lost. In the enumerated list below, the iterative scheme is outlined step-by-step.

1. Choose an initial guess $x^* = \tilde{x}^{1:k_{lost}}$. This could be zero if need be.
2. Form the error equation

$$A_r\left(\bar{x}^{1:k_{lost}} - x^*\right) = \hat{c} - A_r x^* \quad \Rightarrow \quad A_r e = r^* \tag{12.10}$$

   Find the e with minimum euclidean norm $\|e\|_2$ among all admissible solutions.
3. Update $\bar{x}^{1:k_{lost}} = x^* + e$.
4. If $\|e\| \,/\, \|x^*\| < \varepsilon$, algorithm converged.
5. Otherwise, modify the data $\bar{x}^{1:k_{lost}}$ with any a priori knowledge of the structure of the data to recover.
6. Set $x^* = \bar{x}^{1:k_{lost}}$, go to 2.

To test the method, we use it to recover images lost. In the test-case, all 16 images are stored separately in 16 data sets, each image being 512x512 pixels. Another 3 data sets, consisting of checksums, are computed element-wise, per color, using (12.3), to protect the images. The images used are depicted in Figure 12.3 in their original form. (12.9) are used for step 2-3 in the method outlined above.

In Figure 12.4, 4 images have been removed, and then recovered. The approximations recovered are almost indistinguishable to the originals in Figure 12.3. In figure 12.5, 6 images have been removed, i.e. the recovery procedure is attempting to recover having only half as many checksums parity sets as data sets lost. The results are still visually pleasing, although the recovered images have clearly been degraded in quality compared to the originals. In the test, no compressed data was used to initiate the algorithm. As a guess for the first iteration, a zero matrix was used.

An important thing to note about the results presented is that in the encoding procedure, and subsequent decoding procedure, a set of randomly generated checksum matrices A was used in a round-robin fashion, rather than just using a single A repeatedly as is normally the case with erasure codes. Doing so improved the quality of the recovered images substantially compared to using the same matrix A for all elements. In general, our observation was that the increasing the number of randomly generated checksum matrices used would also increase the quality of the images recovered. Using 4, or 16, randomly generated matrices was clearly better than using a single matrix, the improvements were found to be diminishing however, as using 64 matrices instead of 16 would yield results only marginally better.

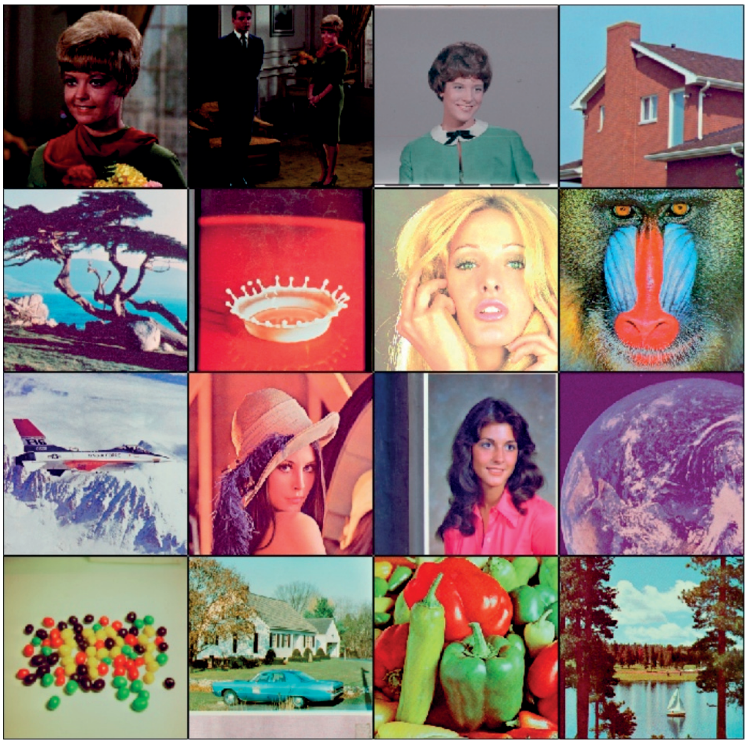
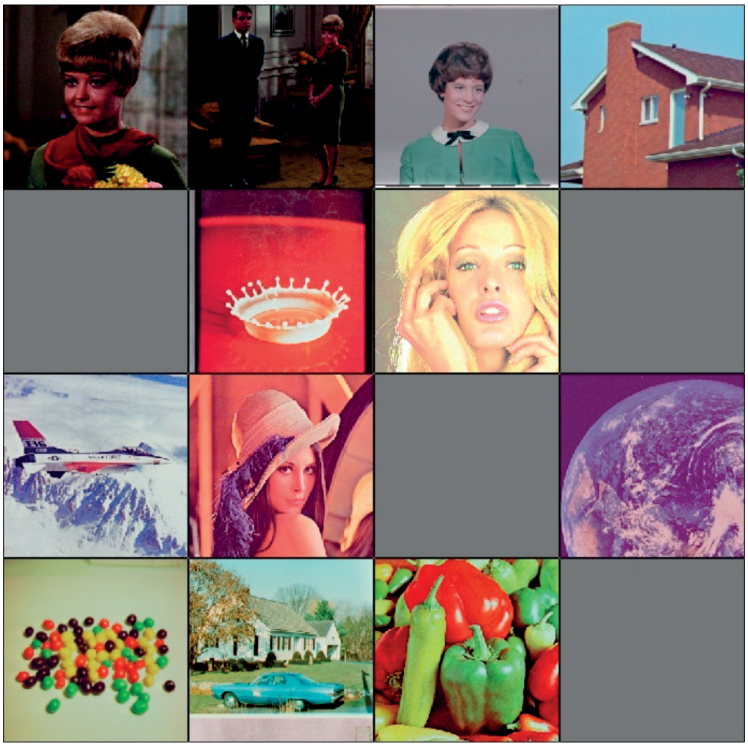
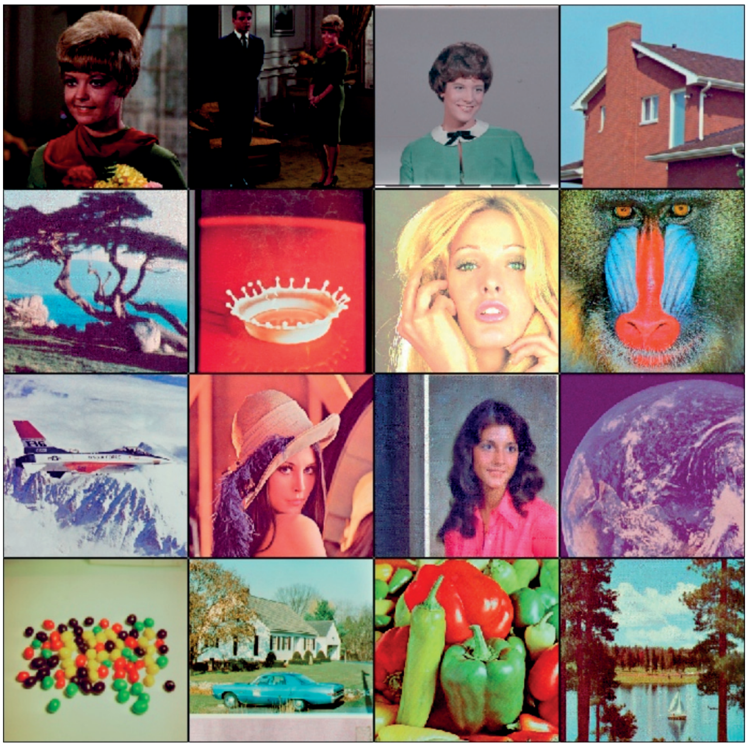


Figure 12.3 – The original version of the 16 images used to demonstrate the method outlined. Each image is 512x512 pixels, and stored in separate data containers that are assumed to be failure prone. The images are taken from the The USC-SIPI Image Database[303]. In figures 12.4 and 12.5, the partial information recovery procedure is demonstrated when removing 4 and 6 images respectively.

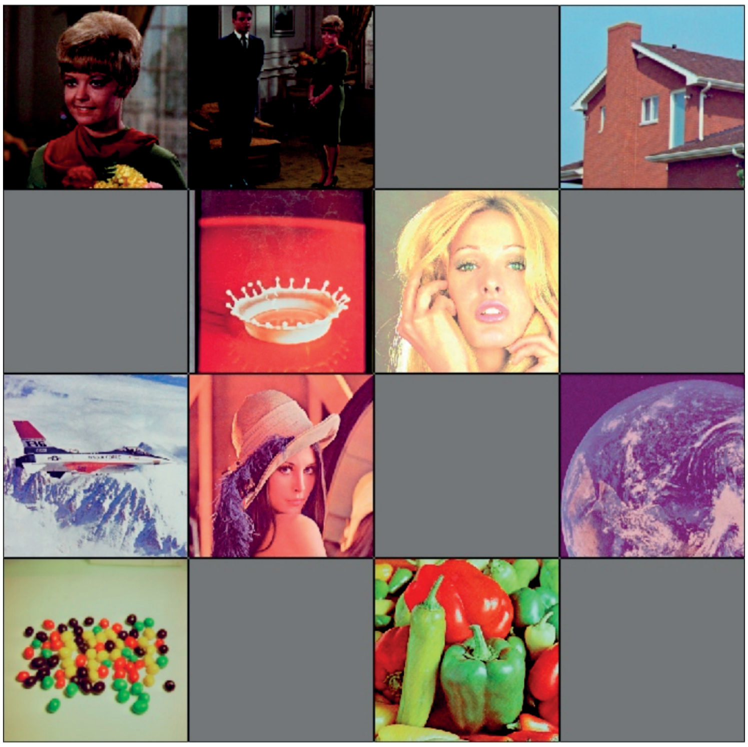


(a) The four images removed.

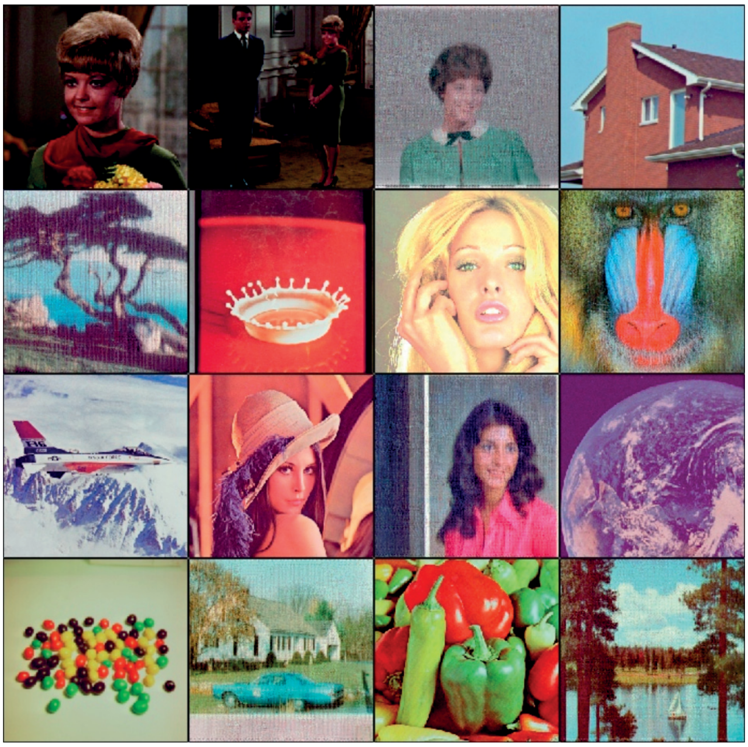


(b) The four images reconstructed.

Figure 12.4 – Three checksums were computed to protect the content of the 16 containers. (a) Shows four images removed. (b) Depicts the recovered images.



(a) The six images removed.



(b) The six images reconstructed.

Figure 12.5 – Three checksums were computed to protect the content of the 16 containers. (a) Shows six images removed. (b) Depicts the recovered images.

12.3 Summary

In this chapter we set out to investigate to what extent it might be possible to find an approximate solution to the problem of recovering lost data from an under-determined checksum system, when having some knowledge of the underlying structure of the data encoded. We proposed a new method, and tested its application on a weighted checksum scheme for floating point numbers. Our preliminary finding is that the answer is yes, it is indeed possible to partially recover the data otherwise considered lost.

The data recovered when $k_{lost} > m$ is not of machine accuracy with respect to the original data, so for the approach to be of practical use in applications like the multi-level checksum checkpointing scheme presented in chapter 11, one would need some way of quantifying the accuracy expected of the data recovered.

The method as presented works best when generating and using multiple checksum matrices. These checksum matrices needs to be stored as well since they are needed in the decoding scheme. The added memory overhead is however very small, in the example given the checksum matrices took up 0.8KB of space compared to 12.6MB for the image data.

Using Gaussian matrices to encode a checksum is somewhat of a niche in the context of fault-tolerance as it really only applies to floating point numbers where the exact bitwise representation is not necessary upon recovery. For the method introduced to have practical relevance, it must be extended to the case where the checksum matrix elements are from a Galois field, i.e. Reed-Solomon codes. There are no obvious reasons to believe that the fundamental idea of alternating between enforcing the constraint of the checkpoint equations, and applying some filter, should not work. In practice though, one need a way of completing step 2-3 in the method as outlined in section 12.2.2. When $A \in \mathbb{R}^{m \times k}$, the Moore–Penrose pseudo inverse could be used as it provides the minimum euclidean norm solution to the under-determined linear system.

A solution procedure for the same problem when $A \in \mathbb{F}^{m \times k}$ is less obvious. Generalized inverses of matrices with elements from finite fields is a research topic that has received limited attention. Some first results on matrices over finite fields that satisfy the four criteria of a Moore-Penrose pseudo inverse were published in [119]. In [310], necessary and sufficient conditions on $A \in \mathbb{F}^{m \times k}$ for the existence of A^{+} was given, and in [78] a method for constructing the Moore–Penrose pseudo inverse was presented. It is however not clear if/how the minimum norm property applies here. In short, further studies are needed to generalize the method.

If successfully extended, the method could potentially have a wide number of applications in fault tolerance and error correction, also outside the context of HPC. Today the average data-center consumes as much power as a small city. In the US alone, data-centers and cloud-service providers store several hundred million terabytes of data, and account for more than 2% of the nations electricty consumption[276]. Overhead related to protection of data accounts for a substantial part of the energy usage, this mainly due to extensive use of redundancy and erasure code to ensure that the probability of hard-drive data loss is extremely small. If a method existed that would allow for robust partial information recovery in the commonly used erasure codes, this would mean that failures that would otherwise result in complete data loss would instead only result in loss of data fidelity which in turn could potentially lead to relaxed requirement on data protection for certain applications.

Summary Part IV

13.4 Novel Contributions

The main contributions of the thesis were presented in Chapters 6 to 8 and Chapters 10 to 12. In each Chapter, issues regarding parallel scaling and fault tolerance were considered. The problems considered all seek to improve parallel efficiency in applications as they scale to larger machines. Novel contributions of the thesis are outlined in the list below

- In Chapters 6 and 7 it was demonstrated that Parareal may be applied for the Parallel-in-Time integration of convection dominated PDE problems under certain conditions. As long as the coarse operator, the preconditioner, only introduce dissipative errors with respect to the fine operator, the algorithm convergences. The findings were used to accelerate a space-parallel finite-volume solver for the nonlinear shallow water wave equation with Parareal. In doing so we devised a new scheduler, denoted Communication Aware Adaptive Parareal (CAAP). With CAAP, we demonstrated a speedup of 228 in the space-time parallel tests, compared to a maximum speedup of 49 for the original parallel-in-space code. In doing so we demonstrated that it is possible to obtain time-parallel speedup for a nonlinear hyperbolic problem in excess of what may be obtained using conventional spatial domain-decomposition techniques alone. These findings offer substantial progress in the understanding of how to move past the strong scaling saturation limit of classical (spatial) domain-decomposition methods for complex convection dominated problems.

- A new method for constructing Parallel-in-Time integration schemes, better suited for hyperbolic and convection dominated problems, is proposed and investigated in Chapter 8. We show than one may construct schemes that work well on simple problems where the classical Parareal scheme otherwise fail. The results indicate that it is possible to construct schemes in which the correction procedure is better suited for problems with strong advective components. However, as with Parareal, the schemes tend to suffer from stability issues when used on more general problems, and as such, the difficulty lies in finding a way to guarantee stability without adversely effecting the speed of convergence.

- In chapter 10, a fault tolerant variant of Parareal was developed and tested. The algorithm exploits clever scheduling, the usage of ULFM-MPI and the interpretation of Parareal as a fixed point iteration for resilience against both hard and soft failures.

- Periodic checkpointing to the parallel-file-system is the workhorse of fault-tolerance in high-performance computing today. This is expected to continue in the foreseeable future, and it has been suggested that extending the approach trough multi-level checkpointing is the most viable approach for minimizing the cost of checkpointing and reducing the compute work lost upon failure[81]. In chapter 11, a C++/MPI library for multi-level checkpointing is presented. The library, called Llama, supports, through ULFM-MPI, automatic rollback without application restart. It is the first library to support both automatic rollback without restart and the use of an arbitrary number of checkpoint levels with topology aware checksum check. Trough numerical experiments, it was demonstrated that the library can achieve in excess of 300GB/s encoding on 256 nodes.

- Erasure codes are widely used to protect data in data centers and are beginning to find usage for fault-tolerance in high-performance computing applications. Data protected through the creation of checksum parity code is, however, only recoverable if the number of lost data and/or code blocks are smaller than, or equal to, the number of checksums parity codes created. In chapter 12, a method for partial information recovery in incomplete checksums was presented. The methods require some knowledge of the underlying structure of the encoded data. Through numerical experiments it was demonstrated that data, otherwise considered lost, may be recovered, at least partially.

In section 13.5 below, we first present a brief overview of current trends and development in high-performance computing, and then discuss perspectives for future work in that context.

13.5 Outlook and Perspectives

World-class supercomputing has become about more than world-class science. Crowning the TOP500 list has become something of a statement, a platform to demonstrate leadership in technological progress and scientific innovation, as well as a source of national pride. After many years of US dominance, the EU, China, and Japan have all fielded large innovative machines in recent years, and each have plans in progress to build an Exascale supercomputer. In 2010, China's HPC presence was almost nonexistent. This has changed dramatically during the past eight years, particularly since HPC development became a key objective in the country's 13'th 5-year plan. From 2015 to 2017, the Gordon Bell prize was won each year by Chinese teams running applications on Chinese supercomputers[93, 209]. In June 2018, the US reclaimed the TOP500 throne with the IBM build Summit supercomputer. The development in HPC is, however, currently moving at a breakneck paste, and the machine might already lose its position on the November 2018 update of the list since several other Exascale prototype machines are under construction.

It has been suggested that the first Exascale systems may be motivated in part by politics. As a result, it could be both unreliable and have unreasonable power requirement, with the aim for it to be operational long enough to run the Linpack benchmark, whereas applications of scientific interest would mostly be unable to scale efficiently to the full size of the machine[17]. It remains an open question as to which country will be the first to field an Exascale machine. After delays in several programs, the US, China, and Japan now all aim to deploy their first Exascale machine during 2021, each country relying on homegrown hardware to claim the honor.

Recent reports would suggest that the Exaflop compute barrier has already been passed by the Summit supercomputer, deployed June 2018 at Oak Ridge National Laboratory[159]. According to these reports, researchers were able to demonstrate 1.88 Exaflops of peak performance on an application for analyzing genomes. The machine derives the bulk of its computational power from 27648 Nvidia Tesla V100 GPUs. This generation of GPUs are equipped with what is called Tensor Cores, special purpose circuits made solely for 4x4 dense matrix-matrix multiplication of mixed precission[204]. Including the special purpose Tensor Cores, the machine has a combined theoretical compute throughput of 3.3 Exaflops. Most applications, however, need to perform other operations than just many small, almost independent, dense matrix-matrix multiplications. The machine achives 122.3 Petaflops of sustained performance on the HPLinpack benchmark, LU factorization of dense matrices[239], and 2.9 Petaflops sustained performance on the HPCG benchmark, preconditioned conjugate gradient for sparse matrices[90]. As to the question of whether or not an Exascale machine already exits? Yes and no, Summit has a theoretical compute throughput that exceeds an Exaflop of mixed precision performance, but in practice this is only achievable on very specific parts of certain applications.

This in turn raises the question of when a computer can really be considered an Exascale machine. Performance of different applications may vary by orders of magnitude, and a machine, suitable for one problem, may be less suitable for other problems. The question therefore has in some sense become meaningless if one does not, at the same time, specify “on what application?”. Perhaps it would be beneficial if the HPC community took a turn of perspective and started focusing on Exascale Applications rather than Exascale machines?

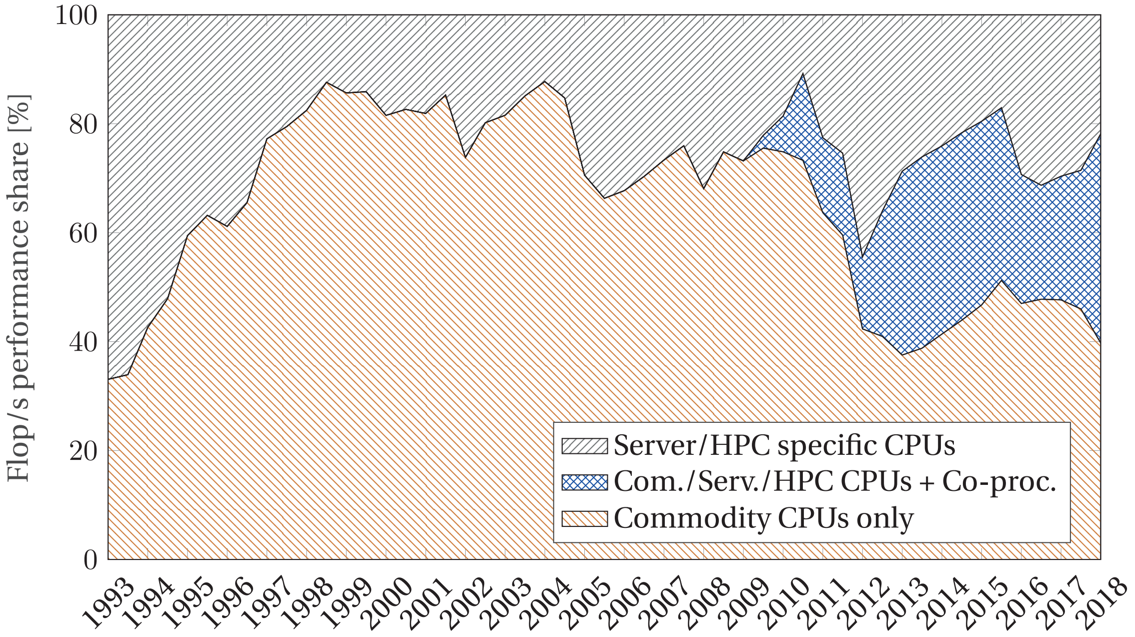
Extreme parallelism and heterogeneous compute architectures have become the norm in large-scale supercomputing as discussed in Part I. A massive investment in new system developments has been seen in recent years as countries prepare for Exascale, and new hardware is being developed at an unprecedented rate. Unfortunately, the very high theoretical compute throughput of the new machines often remain theoretical. The new machines frequently require legacy codes to be rewritten, and sometimes even algorithms to be changed. According to prominent researchers in the field of HPC, there has been an overinvestment in hardware during the past decade and a shift towards funding of scientific software development is needed for the community to fully take advantage of the new machines[264, 93].

In the early 90s, the HPC community experienced a massive shift in technology. New generations of microprocessors, manufactured relatively cheaply in great numbers for the PC and workstation market, offered much better price/performance ratios than vector multiprocessors, specially designed for HPC use. Over the course of two decades, commodity hardware clusters came to almost entirely outcompete hardware made specifically for HPC purposes[291]. Figure 13.6 and 13.7 contains plots of the performance and system share of processors as a function of time since 1993 on the TOP500 list. The change that happened in the early 90s is clearly observed. The figures also indicate that things might be changing in HPC yet again, with a new trend towards greater specialization. Among the world TOP50 computers, the majority of systems are now mostly relying on either HPC specific hardware or a combination of commodity CPUs and compute accelerators.

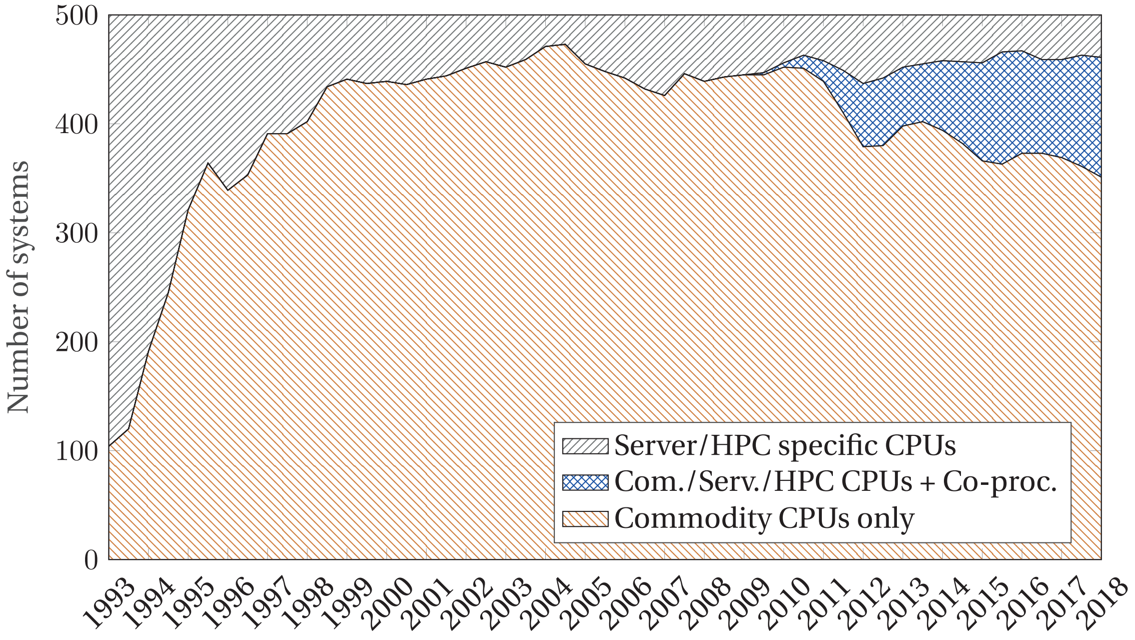
In terms of performance share, more than 50 percent of installed capacity now comes from compute accelerators. Even more HPC centric specialization is on the way, as both the EU, Japan and China plan to use Reduced Instruction Set Computing (RISC) based architecture for processors in their Exascale machines. It appears that the HPC community is moving into a period of greater processor diversity and more specialization after having relied mostly on commodity CPU's for 20 years.

Company	Processor technology
AMD	Athlon, Opteron, x86_64
DEC	Alpha EV4, Alpha EV5, Alpha, EV56 Alpha, EV67, Alpha EV68
Hewlett-Packard	PA-RISC PA-8000, PA-RISC PA-8200, PA-RISC PA-8500, PA-RISC PA-8600, PA-RISC PA-8700
IBM	POWER1, POWER2, POWER3, POWER4, PowerPC 6xx, PowerPC 7xx, PowerPC 9xx
Intel	i860, Pentium, Core, Penryn, Nehalem, IA-32, IA-64, EM64T, Westmere, SandyBridge, IvyBridge, Haswell, Broadwell, Skylake
MIPS Technologies	MIPS R8000, MIPS R10000, MIPS R12000, MIPS R14000, MIPS R16000
Sun Microsystems	SuperSPARC I, UltraSPARC, UltraSPARC II, UltraSPARC III

Table 13.1 – List of processors used in HPC from 1993 to 2018 which was considered commodity processors when creating figures 13.6 and 13.7. The distinction between commodity and non-commodity hardware is not sharp. The above processors were selected because they were all, at some point, used in personal computer workstations, or it was planned for them to be used for such purposes during the design stage.

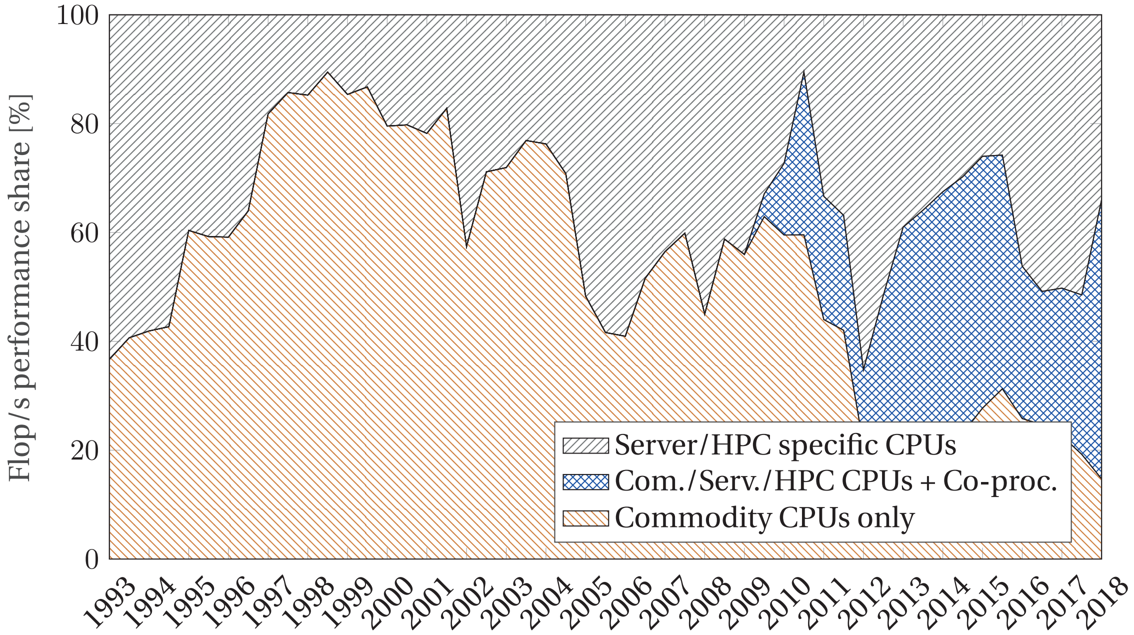


(a) Performance share TOP500

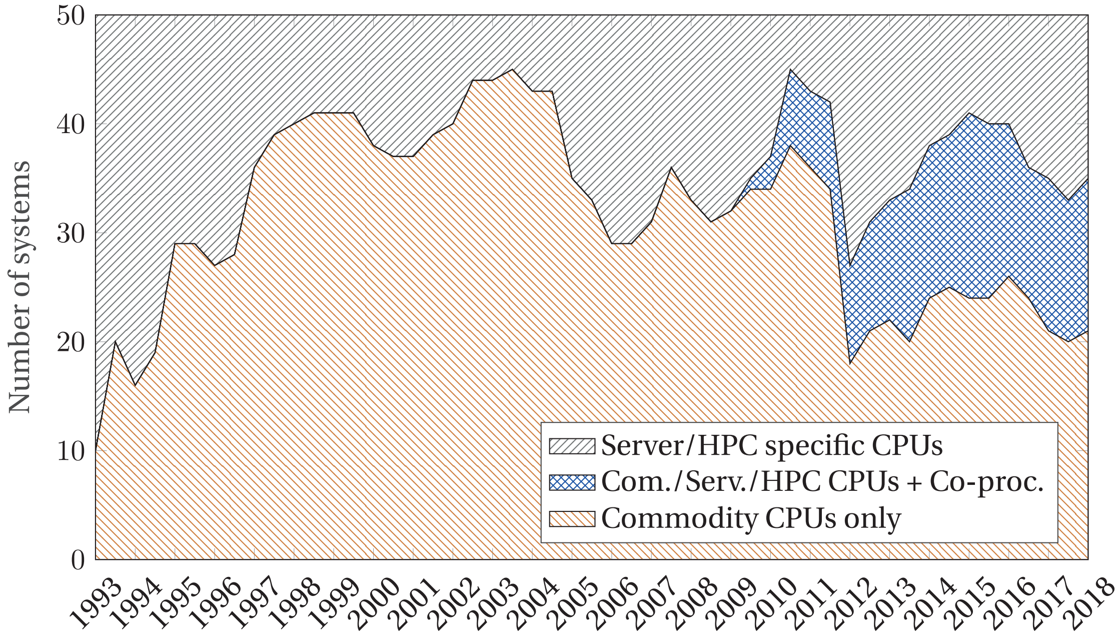


(b) System share TOP500

Figure 13.6 – Number of system and flop/s performance share of processors among the systems recorded on the TOP500 list updated twice a year since 1993[209]. The list of CPU's in table 13.1 was used to distinguish between commodity and non-commodity hardware. Note that the share of systems, not relying only on commodity CPU's, appears to be increasing.



(a) Performance share TOP50



(b) System share TOP50

Figure 13.7 – Number of system and flop/s performance share of processors among the 50 most powerfull systems as recorded on the TOP500 list updated twice a year since 1993[209]. The list of CPU's in Table 13.1 was used to distinguish between commodity and non-commodity hardware. Note that machines using server/HPC specific CPUs and those using co-processors, now has a combined 80% performance share among the 50 most powerful systems.

The research presented in the thesis revolve around parallel scalability and fault tolerance for numerical algorithms on large supercomputers. Fundamental laws of physics limits frequency scaling. The way forward towards higher compute performance and energy efficiency has now become, and continues to be, by increasing the number of processors. The IBM Summit supercomputer has a total of 140 million CUDA cores on 2.3 million Streaming Multiprocessors. Specialization of certain circuits will not make parallelism go away, so the challenge of coordination and communication between millions of cores on upwards of 50.000 separate compute nodes will therefore continue in the foreseeable future. Parallel-in-Time integration enable numerical algorithms for solving PDEs to continue scaling across the strong scaling barrier of classical methods, the method continues to be of relevance and may find practical use in certain applications where time-to-solution is essential.

The work presented in this thesis has served to elucidate the difficulties that arise when applying Parareal to convection dominated and hyperbolic problems. In chapter 7, it was demonstrated how the method can be successfully applied to conservation laws solved using certain types of discretizations. Parareal is however severely limited in some ways; dispersive difference of $\mathcal{G}_{\Delta T}$ with respect to $\mathcal{F}_{\Delta T}$ are amplified rather than corrected. We can therefore never expect Parareal to be a good general-purpose approach for convection dominated problems, and new methods for parallel-in-time integration better suited for those types of problems should therefore be developed. In chapter 8, a general approach for deriving new parallel-in-time integration schemes was proposed. Several methods were derived and tested. Although they outperformed Parareal in terms of convergence speed on certain problems, these methods too suffer from issues of stability. Future work should focus on finding a way to derive methods that are guaranteed to converge monotonously so to accommodate better scaling for a wider range of problems.

Tolerance towards hardware-failures was another key issue considered. In Chapter 12, a method for partial information recovery with incomplete checksums was proposed. The number of data-centers around the world has been growing exponentially in recent years[276], and the trend is expected to continue, so such a method could therefore be of potential interest in the industry. The method was, however, derived for use on data represented as real numbers. Most erasure codes, such as the Reed-Solomon algorithm, are for use on data treated as finite field elements, i.e., binary numbers. For the method to have practical relevance, it must be extended from the field of real numbers to the finite field. Likewise, all tests were performed using real numbers representing lines and 2D surfaces, data with well defined structure. Further studies should test how well the method works when the data to be protected contains objects of less well defined structure such as text files or compressed images.

When it comes to algorithm based fault-tolerance in the context of HPC, the future is less certain. In position papers, published in the years 2014-2015, it was conjectured that applications would be unable to run on Exascale machines without the development and deployment of new forms of fault-tolerance[53]. The essential argument was that HPC is a scavengers field, using and adapting commodity hardware. The market for commodity processors has become driven by mobile devices and laptops, which are cost and energy sensitive, but do not require high reliability. It has been estimated that it would be possible to avoid an increase in the frequency of errors, using commodity hardware at Exascale, at the expense of 20% more circuits and energy consumption[282]. This is, however, implausible as in commodity hardware there is no need for such extreme reliability. Thus, it is theorized that if an Exascale machine is to be build of commodity hardware, new methods of fault-tolerance will be essential for any applications using the machine.

Plans for the construction of Exascale machines around the world are now beginning to take form, and it turns out that none of the them will be based on commodity hardware. Furthermore, a recent study on the long-term failure rate of several large clusters of different generations found that the MTBF has remained roughly constant[146]. Newer world-class systems are not more or less reliable than older systems, and the MTBF for these machines continues to be in the range of 8-24 hours. As such, algorithm based fault-tolerance in HPC will probably continue, in the foreseeable future, to be considered a nice-to-have, rather than becoming a must-have as previously thought.

With planned Exascale machines on the horizon, some researchers have begun speculating what lies ahead. The move from Teraflops and Petaflops to Exasflops has seen the introduction of heterogeneous architectures, specialization and extreme parallelism. Prominent researcher have postulated that developing machines capable of Zettaflops (10^{21}) would require changes in hardware even more disruptive than those seen over the course of the past decade[110]. Some even suggest that we will never reach Zettaflops using conventional approaches, and that the community must eventually look beyond the Von Neumann architecture, of which almost all commercial computing systems of the past seventy years have derived from. Optical[174], Neuromorphic[218], and Quantum[219] computing are among the proposed alternatives, but it is still much too early to make any meaningful prediction on which technology will prevail.

As a closing remark to end the thesis, we will quote Prof. Thomas Sterling, recipient of the Gordon Bell Prize and known as the father of Beowulf clusters. He recently wrote on the coming generation of Exascale machines[110],

> "... it is a beachhead on the forefront of nanoscale enabling technologies, marking the end of Moore's Law, the flatlining of clock rates due to power considerations, and the limitations of clock rate. The achievement of exascale computing will serve as an inflection point at which change from conventional means is not only inevitable but essential."

Appendix Part

A CAAP Scheduler on Test Eq.

Appendix A contains schematic drawings of the schedulers presented in chapter 7 when applied to solve the test ODE 7.30 as described in section 7.3 with pauses to replace compute work. Unlike the conceptual drawings used to outline the algorithm in section 7.3, all drawings of the work scheduling in the appendix are made from timings made during the parallel solution procedure and the dumped to a file at the end. The four figures containing a schematic representation of computation, and communication of time-subdomain interface states, are

Figure A.1 The "Fully distributed" parareal scheduler as proposed in [9].

Figure A.2 Multiple consecutive executions of the fully distributed Parareal.

Figure A.3 CAAP. Here using with $\beta = 0$ so that node-groups that finished a time-subdomain in a cycle will wait for a correction to be made before receiving a new state to commence their work.

Figure A.4 CAAP. Here using $\beta = 1$ so that node-groups that finished a time-subdomain in a cycle will receive a new state to commence their work immediately.

In all figures, blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed receive of solution state and convergence flag. Dark gray indicates $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$ computation in progress, light gray indicates $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$ being computed. For CAAP where multiple cycles of Parareal overlap in execution time so that otherwise idle node-groups commence their work on the following cycle before the current cycle has completed, the green squares and arrows indicate signal flag send and receive.

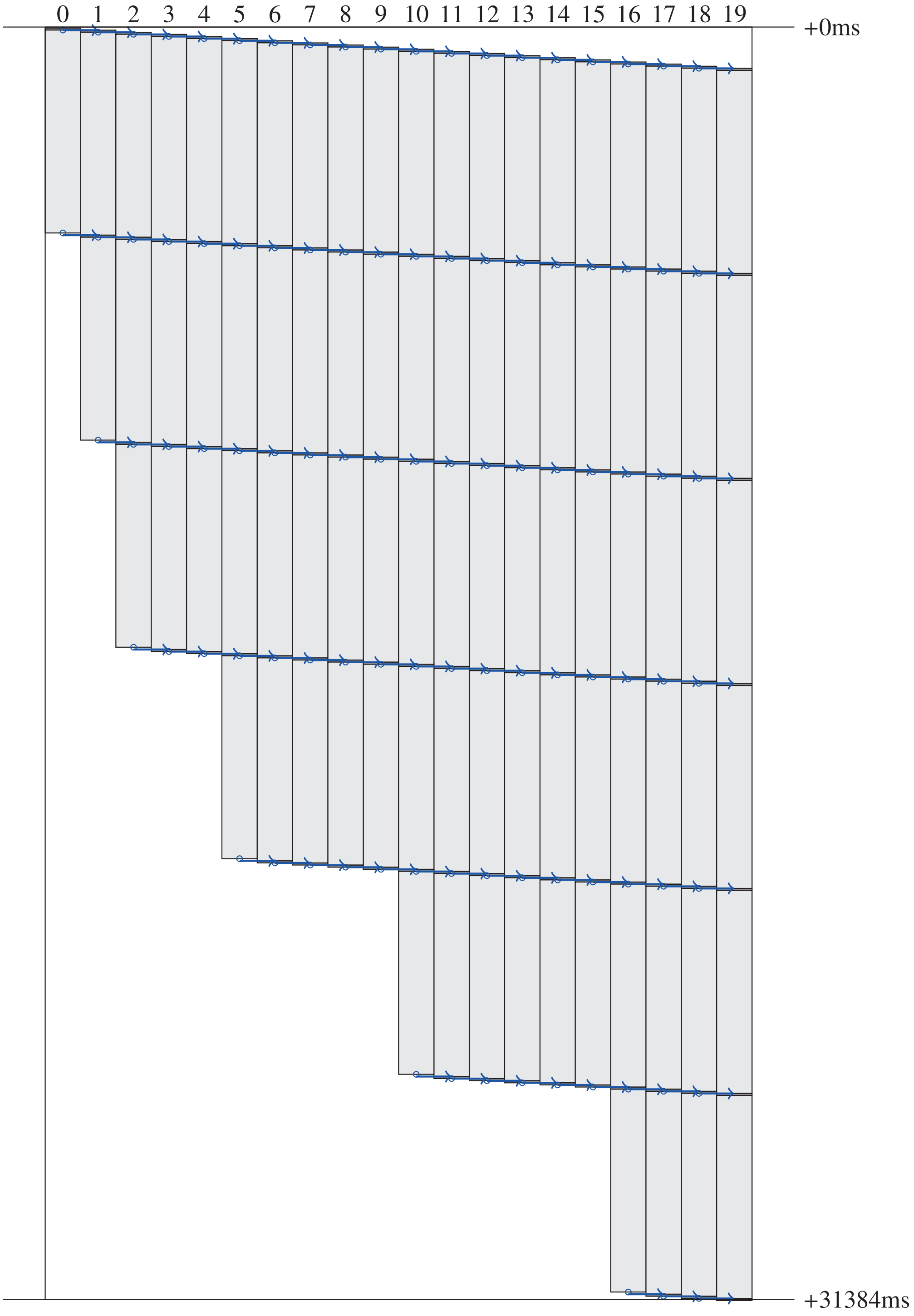


Figure A.1 – "Fully distributed" Parareal. $n_t = 20$, $n_c = 1$, $T = 100$, $\Delta T = 5$, $dT = 10^{-3}$, , $dt = 10^{-5}$, $T_{\mathcal{F}} = 5000ms$, $T_{\mathcal{G}} = 50ms$, $T_{\mathcal{C}} = 2ms$. Blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed receive of solution state and convergence flag. Dark gray indicates $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$ computation, light gray indicates $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$. Measured speedup 3.19, parallel efficiency 15.9%.

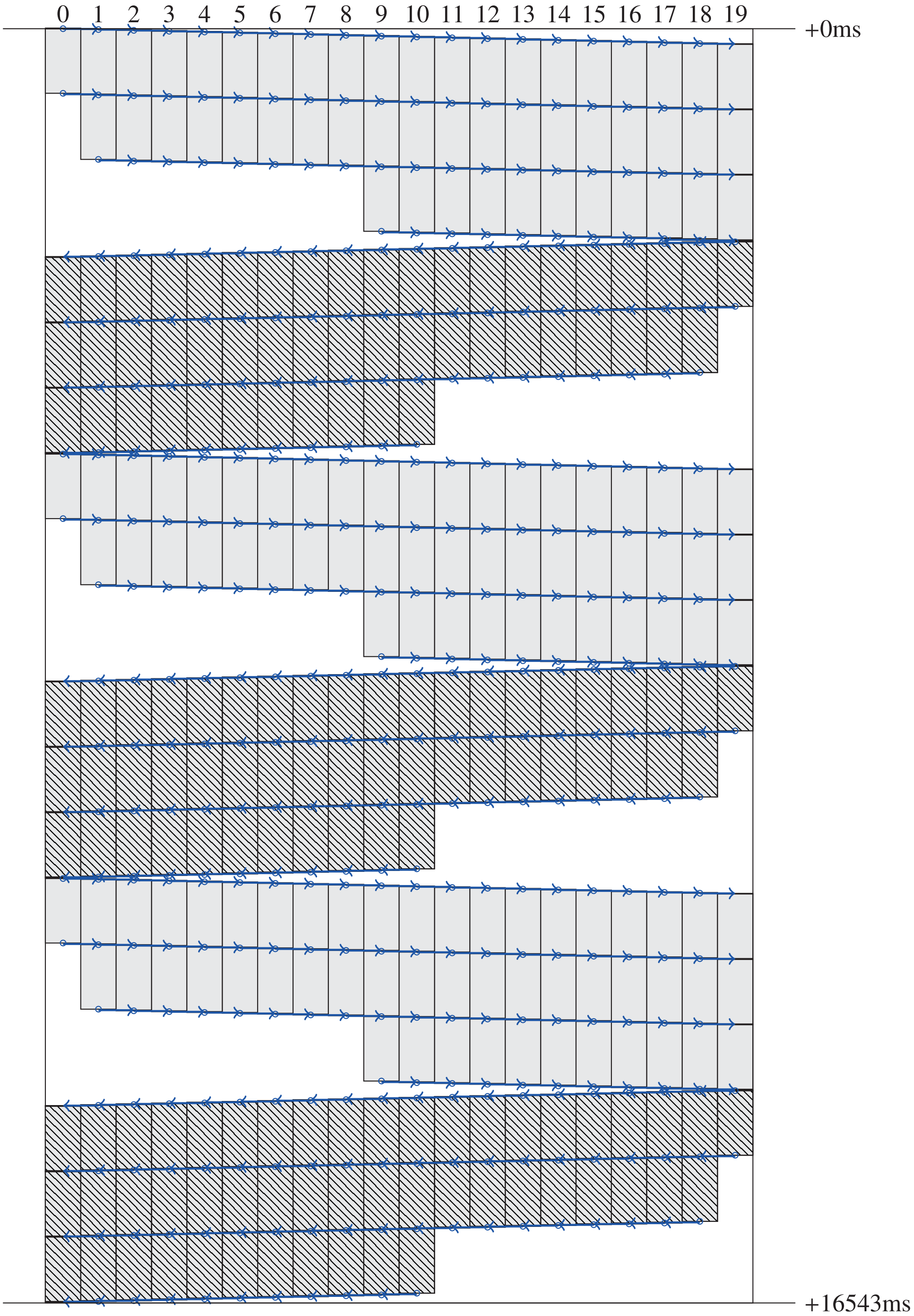


Figure A.2 – Multiple consecutive executions of the standard "fully distributed" Parareal. $n_t = 20$, $n_c = 6$, $T = 100$, $\Delta T = 0.83\bar{3}$, $dT = 10^{-3}$, $dt = 10^{-5}$, $T_{\mathcal{F}} = 4167ms$, $T_{\mathcal{G}} = 42ms$, $T_{\mathcal{C}} = 2ms$. Blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed recieve of solution state and convergence flag. Dark gray computation of $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$. Light gray, and light gray with lines, computation of $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$. Measured speedup 6.04, parallel efficiency 30.2%.

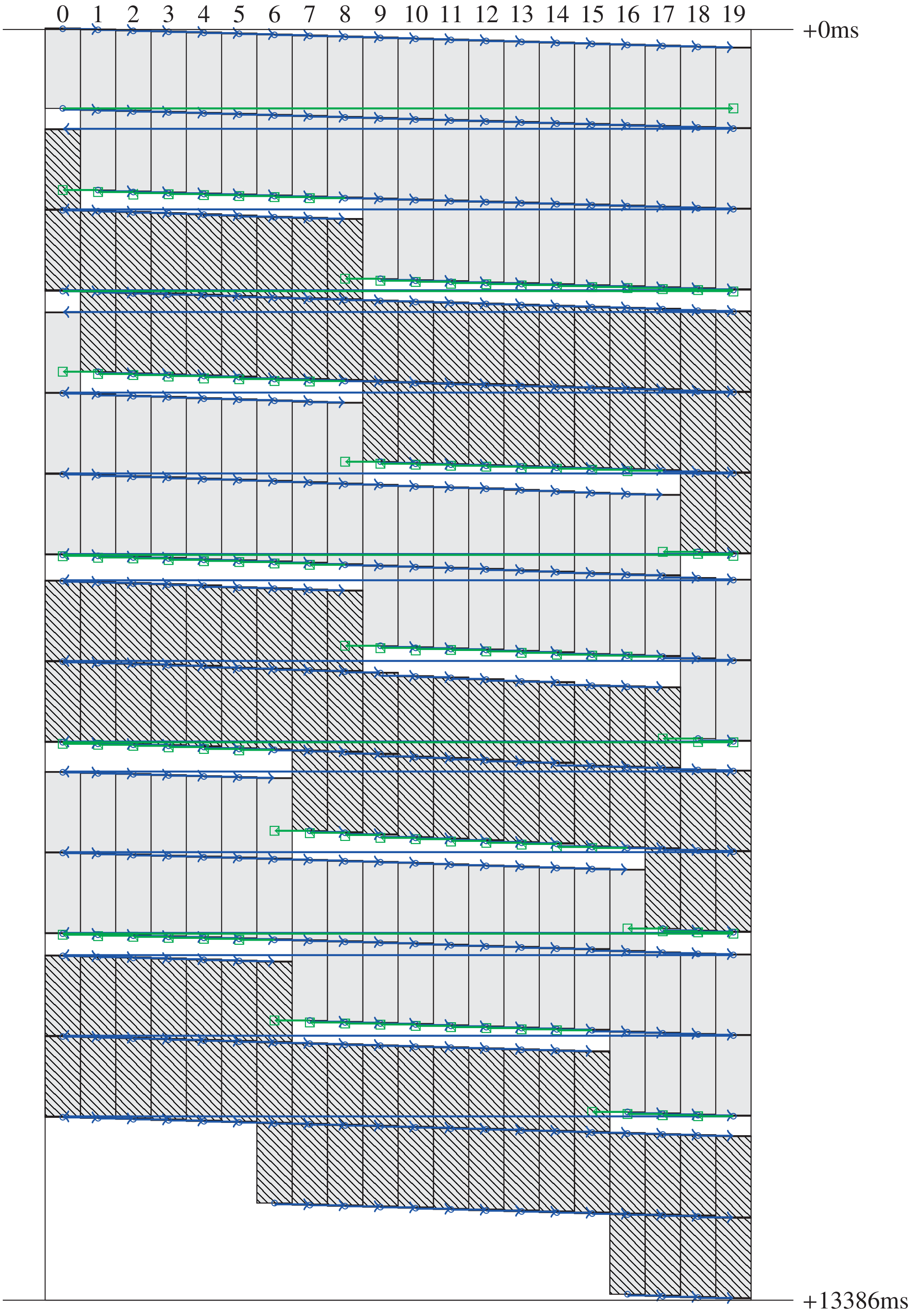


Figure A.3 – Adaptive Parareal Scheduler. $\beta = 0$, $n_t = 20$, $n_c = 6$, $T = 100$, $\Delta T = 0.83\bar{3}$, $dT = 10^{-3}$, $dt = 10^{-5}$, $T_{\mathcal{F}} = 4167ms$, $T_{\mathcal{G}} = 42ms$, $T_{\mathcal{C}} = 2ms$. Blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed receive of solution state and convergence flag. Green square and arrow indicates signal flag. Dark gray computation of $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$. Light gray, and light gray with lines, computation of $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$. Measured speedup 7.73, parallel efficiency 37.4%.

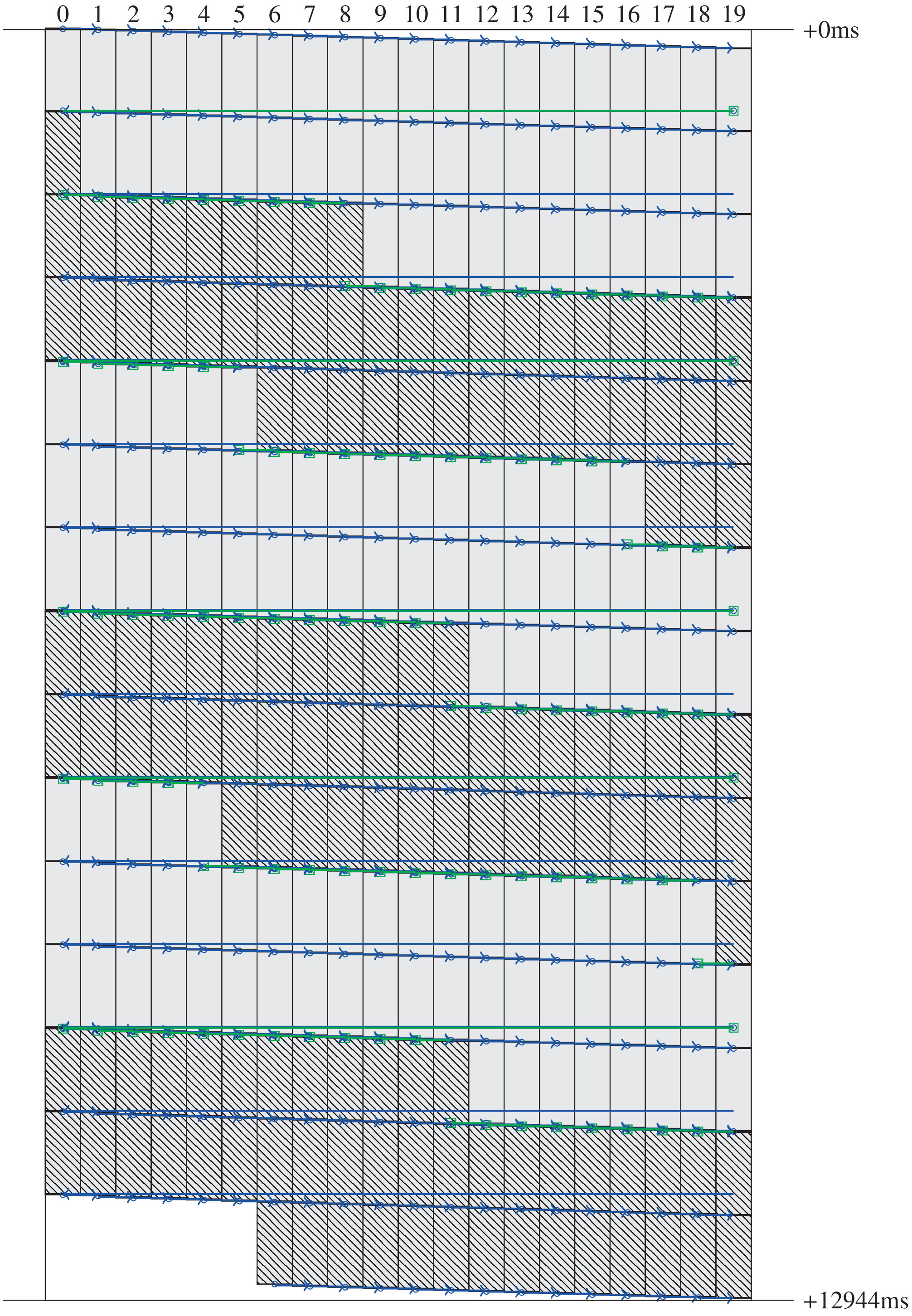


Figure A.4 – Adaptive Parareal Scheduler. $\beta = 1$, $n_t = 20$, $n_c = 6$, $T = 100$, $\Delta T = 0.83\bar{3}$, $dT = 10^{-3}$, $dt = 10^{-5}$, $T_{\mathcal{F}} = 4167ms$, $T_{\mathcal{G}} = 42ms$, $T_{\mathcal{C}} = 2ms$. Blue circles indicate a posted send of solution state and convergence flag. Blue arrows indicate a completed receive of solution state and convergence flag. Green square and arrow indicates signal flag. Dark gray computation of $\mathcal{G}_{\Delta T}\mathrm{U}_n^k$. Light gray, and light gray with lines, computation of $\mathcal{F}_{\Delta T}\mathrm{U}_n^k$. Measured speedup 7.73, parallel efficiency 38.6%.

B CAAP Convergence on Test Eq.

The appendix contains 8 figures investigating the impact on the convergence pattern of the schedulers of three different parameters. Varying the accuracy of the coarse operator δT, the length of the timedomain T, and the tolerance on the norm ϵ between two consecutive iterations used to determine if a time-subdomain should be accepted as having converged. The impact on the convergence pattern of both regular Parareal and CAAP on the test equation 7.30 as described in section 7.3 is investigated. Below a list of figures in the appendix.

Figure B.1 Parareal, $N_c = 1$, $N_p = 10$, $T = 10$, $\epsilon = 10^{-3}$

Figure B.2 Parareal, $N_c = 1$, $N_p = 10$, $T = 100$, $\epsilon = 10^{-2}$

Figure B.3 Multipe Consecutive Parareal, $N_c = 5$, $N_p = 10$, $T = 10$, $\epsilon = 10^{-3}$

Figure B.4 Multipe Consecutive Parareal, $N_c = 5$, $N_p = 10$, $T = 100$, $\epsilon = 10^{-2}$

Figure B.5 CAAP, $\beta = 0$ on $N_c = 5$, $N_p = 10$, $T = 10$, $\epsilon = 10^{-3}$

Figure B.6 CAAP, $\beta = 0$ on $N_c = 5$, $N_p = 10$, $T = 100$, $\epsilon = 10^{-2}$

Figure B.7 CAAP, $\beta = 0.8$ on $N_c = 5$, $N_p = 10$, $T = 10$, $\epsilon = 10^{-3}$

Figure B.8 CAAP, $\beta = 0.8$ on $N_c = 5$, $N_p = 10$, $T = 100$, $\epsilon = 10^{-2}$

Standard single cycle Parareal on $N_c = 1$**,** $N_p = 10$**,** $T = 10$**,** $\epsilon = 10^{-3}$

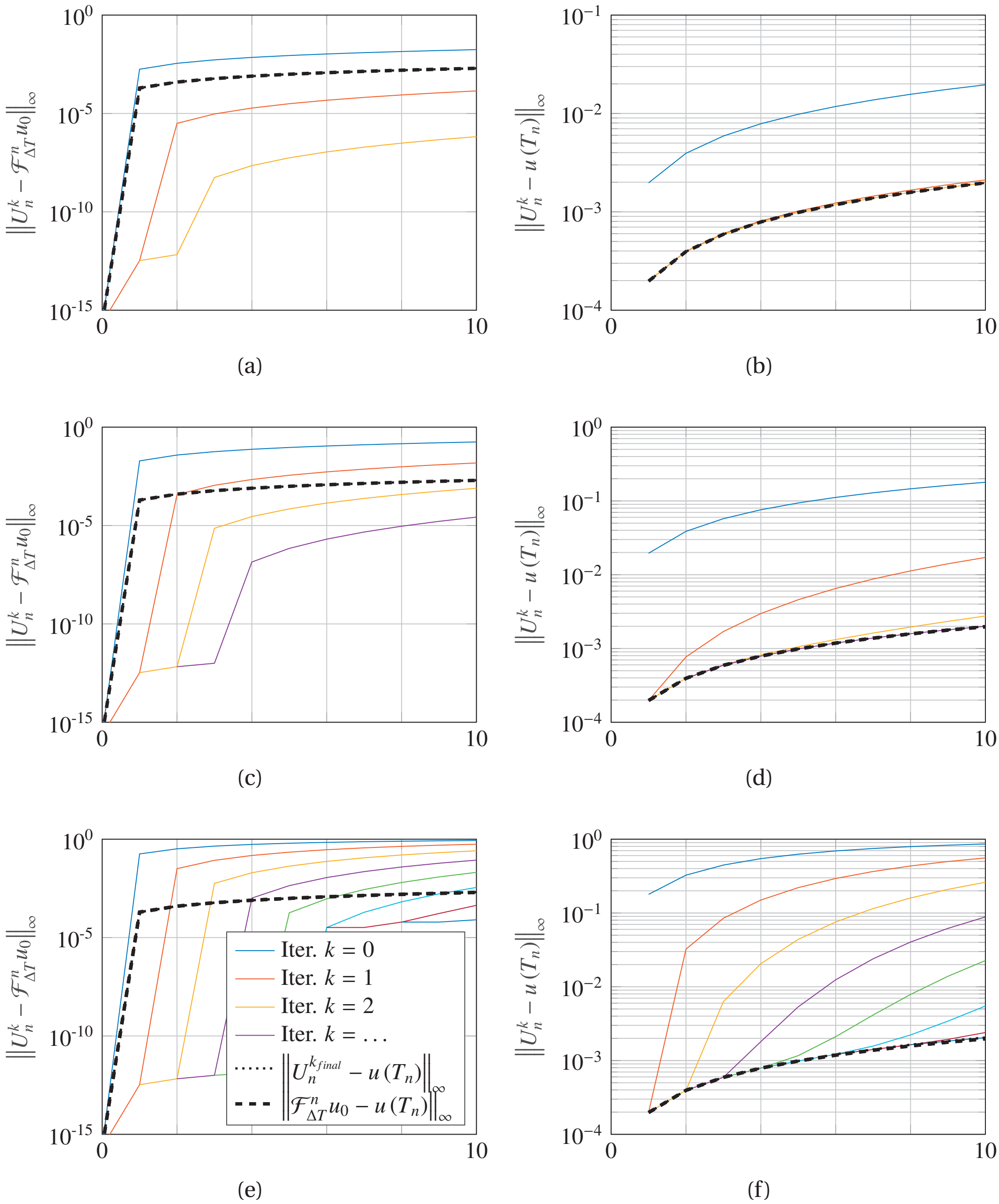
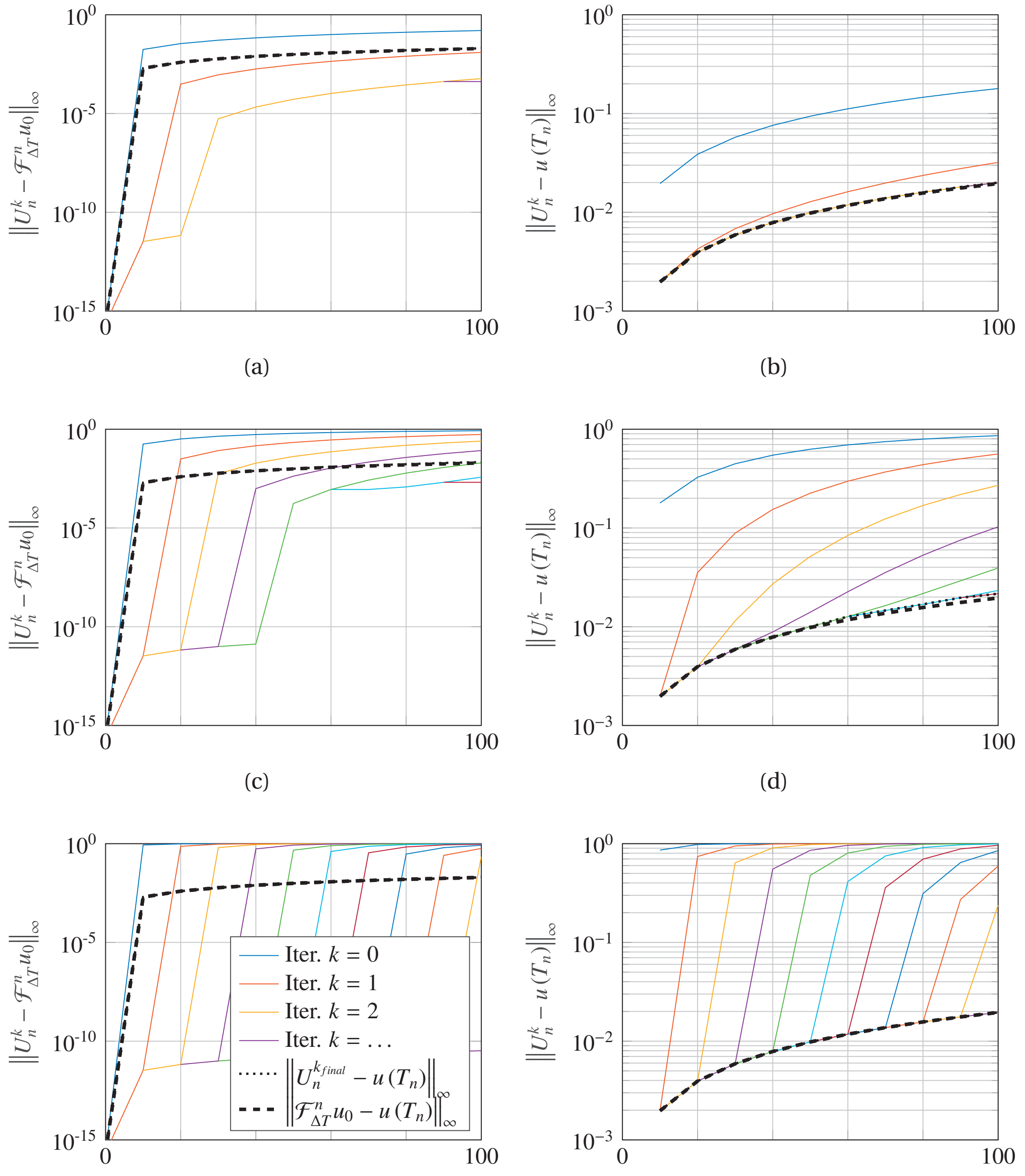


Figure B.1 – Error as a function of simulation time when using standard single-cycle Parareal to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 10$ is performed across $N_c = 1$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-3}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f).

Standard single cycle Parareal on $N_c = 1$**,** $N_p = 10$**,** $T = 100$**,** $\epsilon = 10^{-2}$



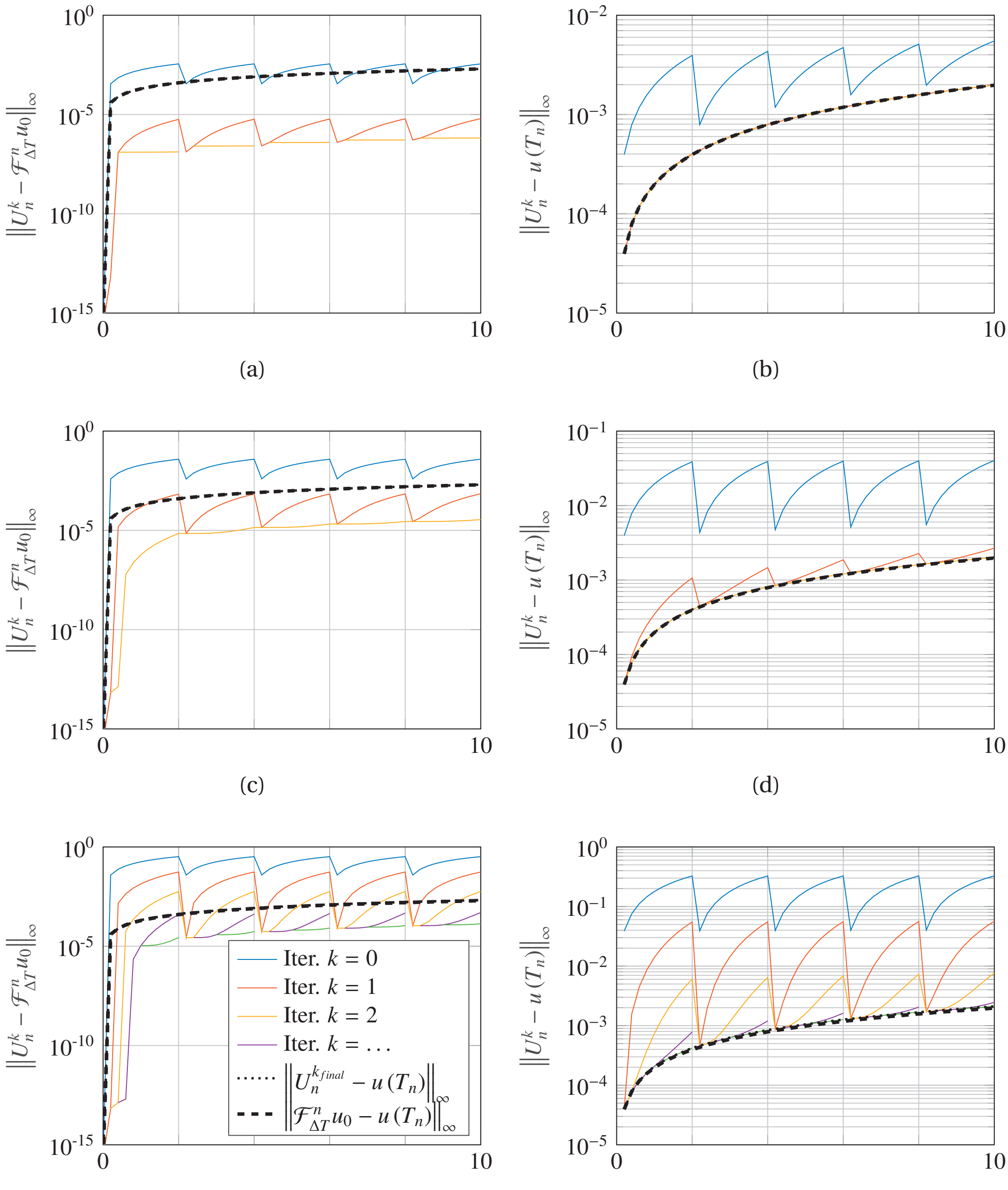
(a) (b)

(c) (d)

(e) (f)

Figure B.2 – Error as a function of simulation time when using standard single-cycle Parareal to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 100$ is performed across $N_c = 1$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-2}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f).

Standard multi cycle Parareal on $N_c = 5$, $N_p = 10$, $T = 10$, $\epsilon = 10^{-3}$



(a) (b)

(c) (d)

(e) (f)

Figure B.3 – Error as a function of simulation time when using standard multiple-cycle Parareal to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 10$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-3}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f).

Standard multi cycle Parareal on $N_c = 5$, $N_p = 10$, $T = 100$, $\epsilon = 10^{-2}$

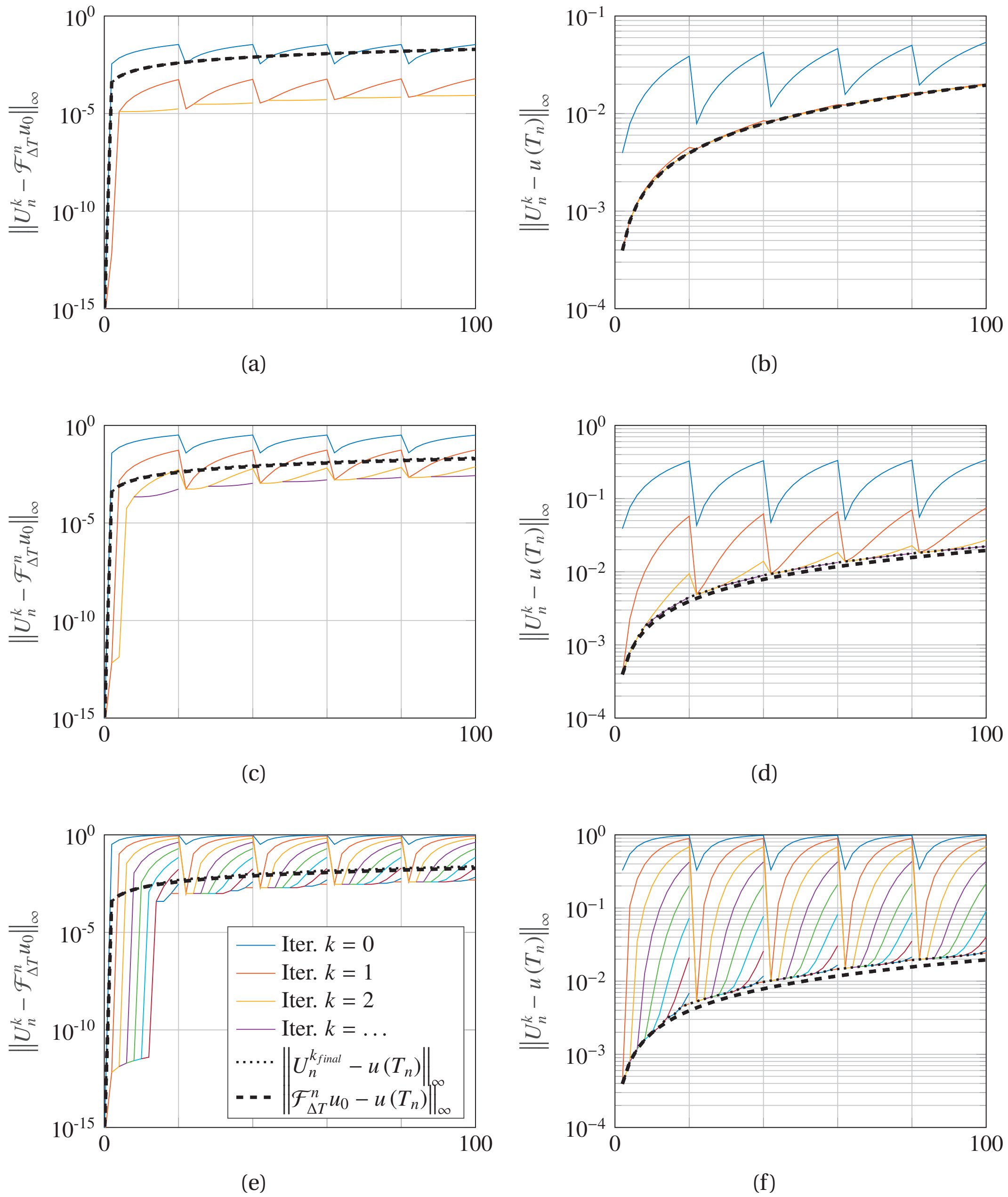
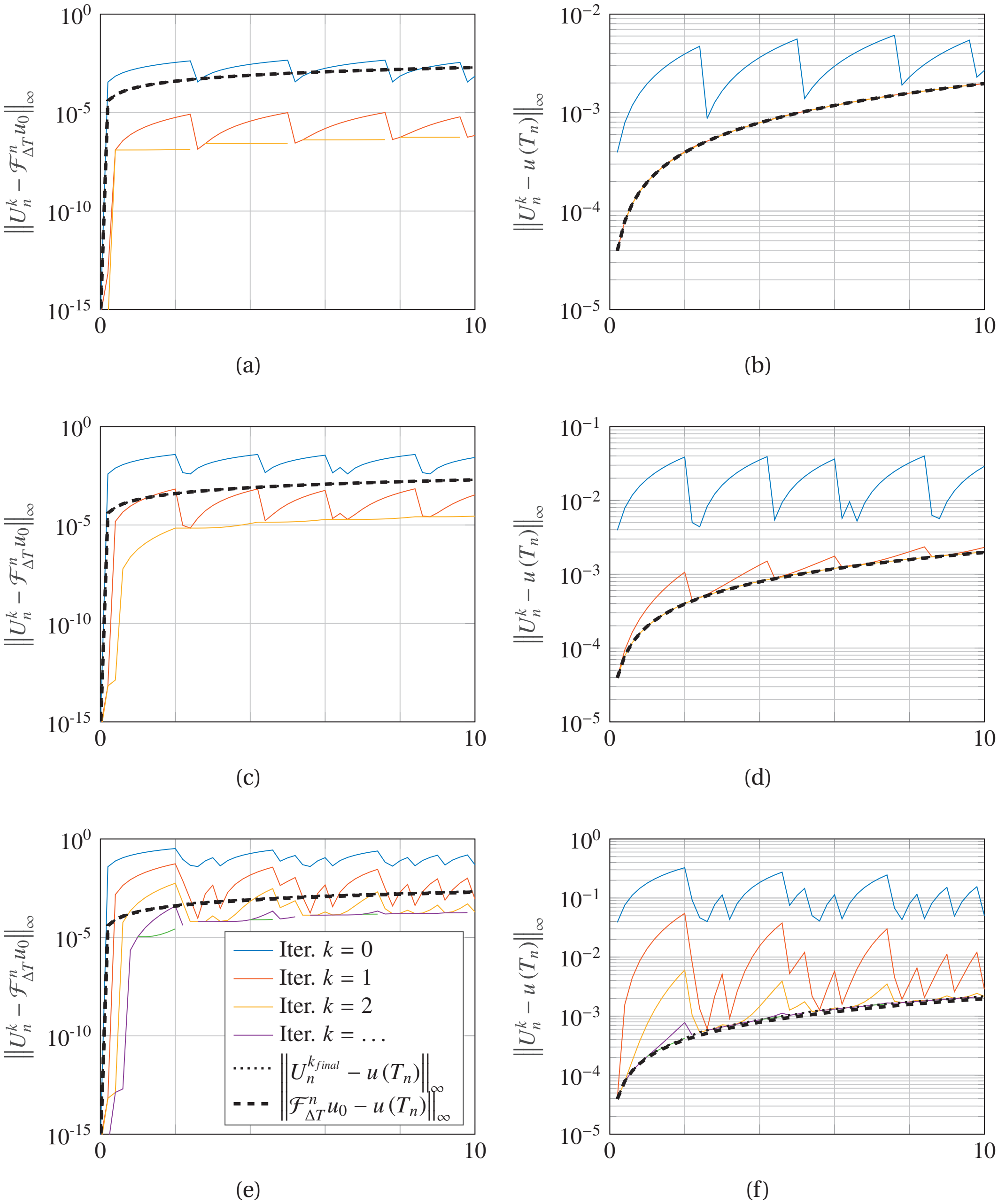


Figure B.4 – Error as a function of simulation time when using standard multiple-cycle Parareal to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 100$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-2}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f).

Adaptive Parareal with $\beta = 0$ on $N_c = 5$, $N_p = 10$, $T = 10$, $\epsilon = 10^{-3}$



(a) (b)

(c) (d)

(e) (f)

Figure B.5 – Error as a function of simulation time when using Adaptive Parareal with $\beta = 0$ to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 10$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-3}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f).

Adaptive Parareal with $\beta = 0$ on $N_c = 5$, $N_p = 10$, $T = 100$, $\epsilon = 10^{-2}$

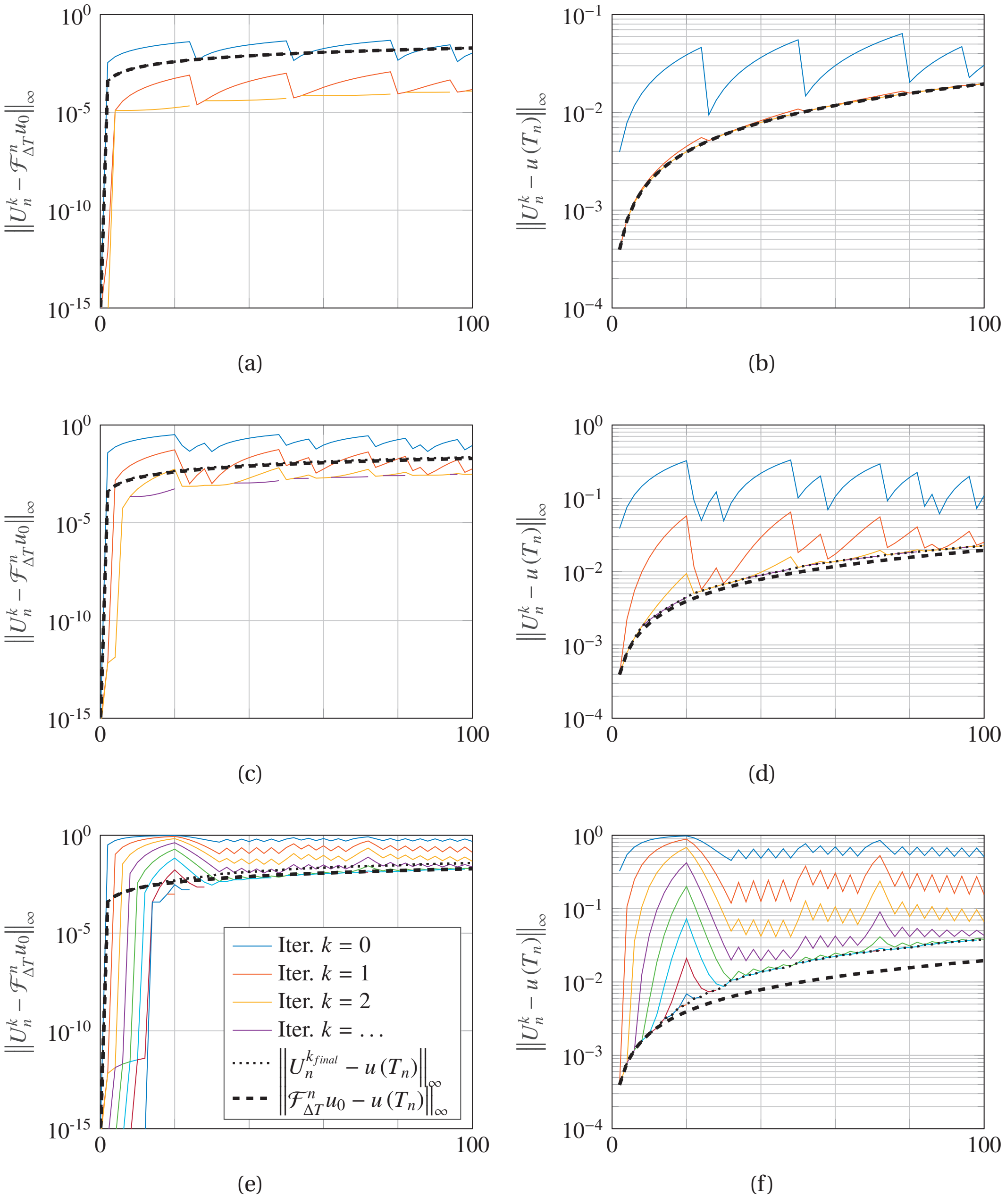
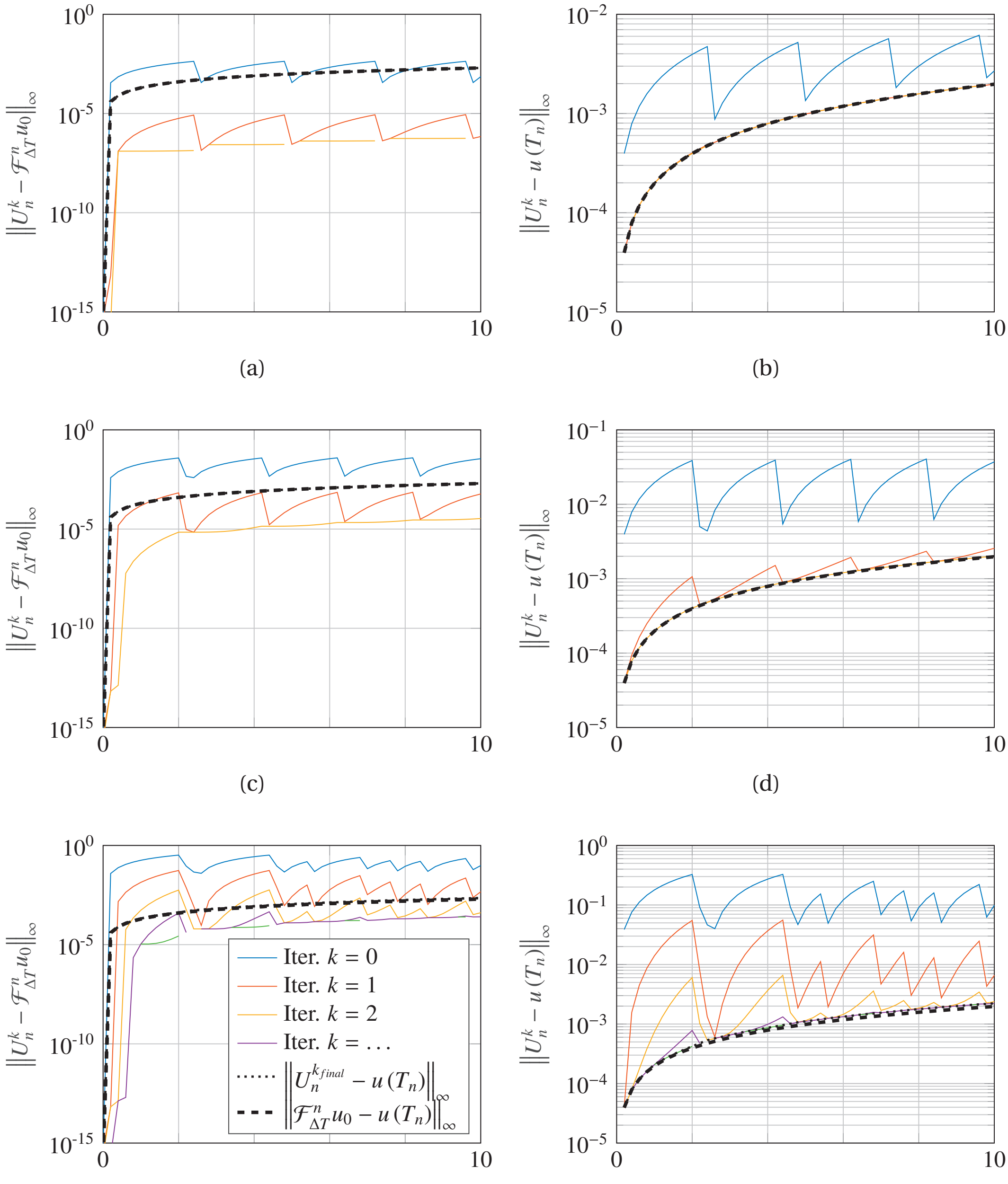


Figure B.6 – Error as a function of simulation time when using Adaptive Parareal with $\beta = 0$ to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 100$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-2}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f).

Adaptive Parareal with $\beta = 0.8$ on $N_c = 5$, $N_p = 10$, $T = 10$, $\epsilon = 10^{-3}$



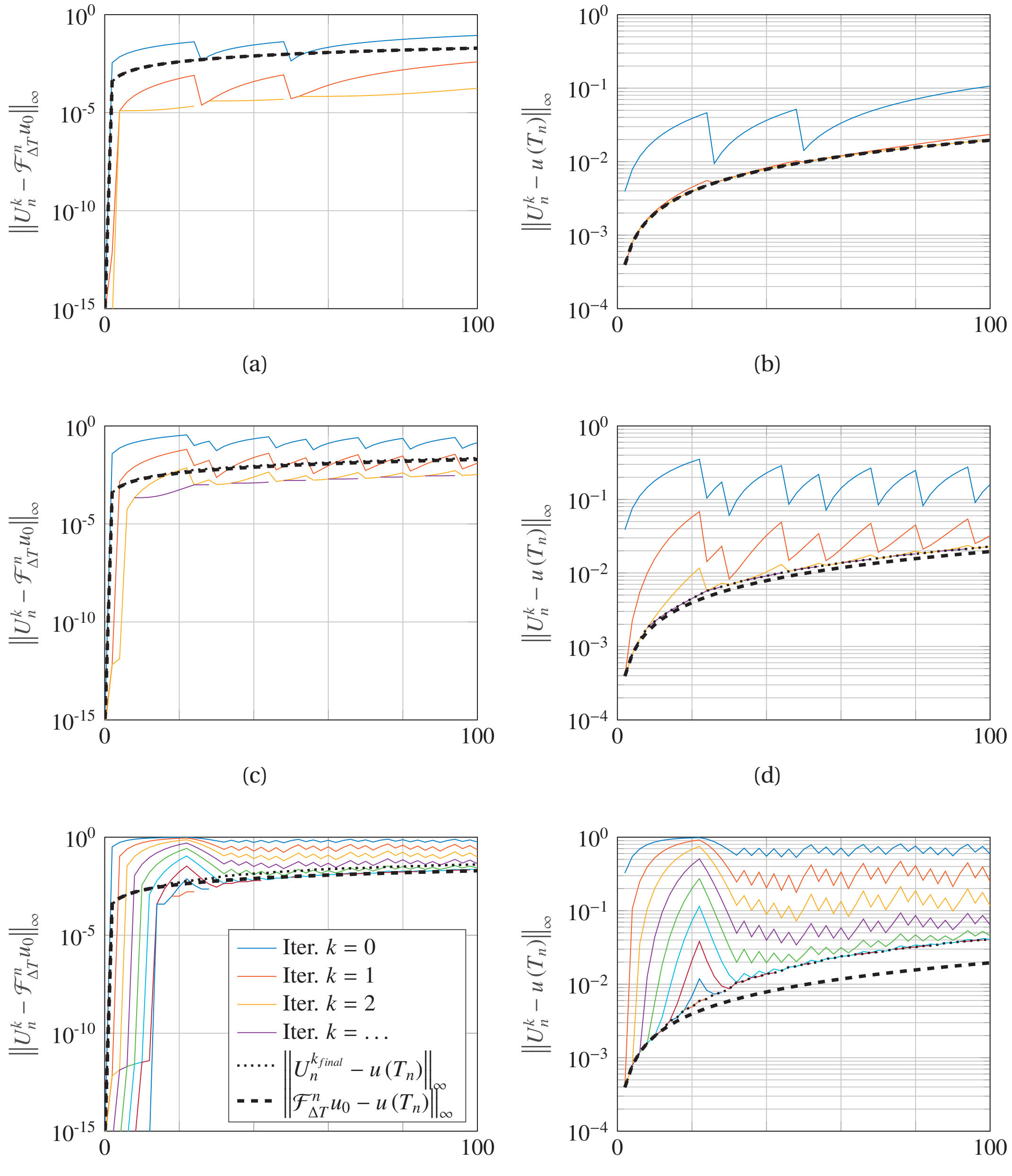
(a) (b)

(c) (d)

(e) (f)

Figure B.7 – Error as a function of simulation time when using Adaptive Parareal with $\beta = 0.8$ to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 10$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-3}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f).

Adaptive Parareal with $\beta = 0.8$ **on** $N_c = 5$**,** $N_p = 10$**,** $T = 100$**,** $\epsilon = 10^{-2}$



(a) (b)

(c) (d)

(e) (f)

Figure B.8 – Error as a function of simulation time when using Adaptive Parareal with $\beta = 0.8$ to integrate equation 7.30. In figures (a,c,e) error is measured with respect to the sequential solution using $\mathcal{F}_{\delta T}$. In (b,d,e) error is measured with respect to the analytical solution. Integration to $T = 100$ is performed across $N_c = 5$ cycles, each cycle with $N_p = 10$ time-subdomains. Tolerance set to $\epsilon = 10^{-2}$. Time-step length in $\mathcal{F}_{\delta T}$ is $dt = 10^{-5}$. Timestep length in $\mathcal{G}_{\delta T}$ is $dT = 10^{-4}$ for figures (a,b). $dT = 10^{-3}$ in figures (c,d). $dT = 10^{-2}$ in figures (e,f).

C CAAP Subdomain length for Optimal Speedup

Appendix C contains figures, for Chapter 7, of measured parallel speedup and efficiency as a function of both time-subdomain length and number of cycles when solving the test equation (7.44). Measurements are presented when using multiple consecutive cycles of the standard fully distributed Parareal scheduler as well as CAAP for $\beta = 0.0$, $\beta = 0.5$, and $\beta = 1.0$. Figures are made so to evaluate how well the a priori estimate for the optimal time-subdomain length δT using equation (7.44) and (7.51) approximate the actual optimal time-subdomain length.

For the numerical experiments, results are presented as a function of both parallel efficiency and of speedup. Experiments are made for three different costs of communicating a solution state, $T_c^w = 1ms, 10ms, 100ms$, and three different coarse operator time-step lengths. The cost of performing the coarse operator $C_\mathcal{G}^w$ is made proportional to it's time-step length.

Figure C.1 Number of cycles N_c for $T = 100$, $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 1ms$

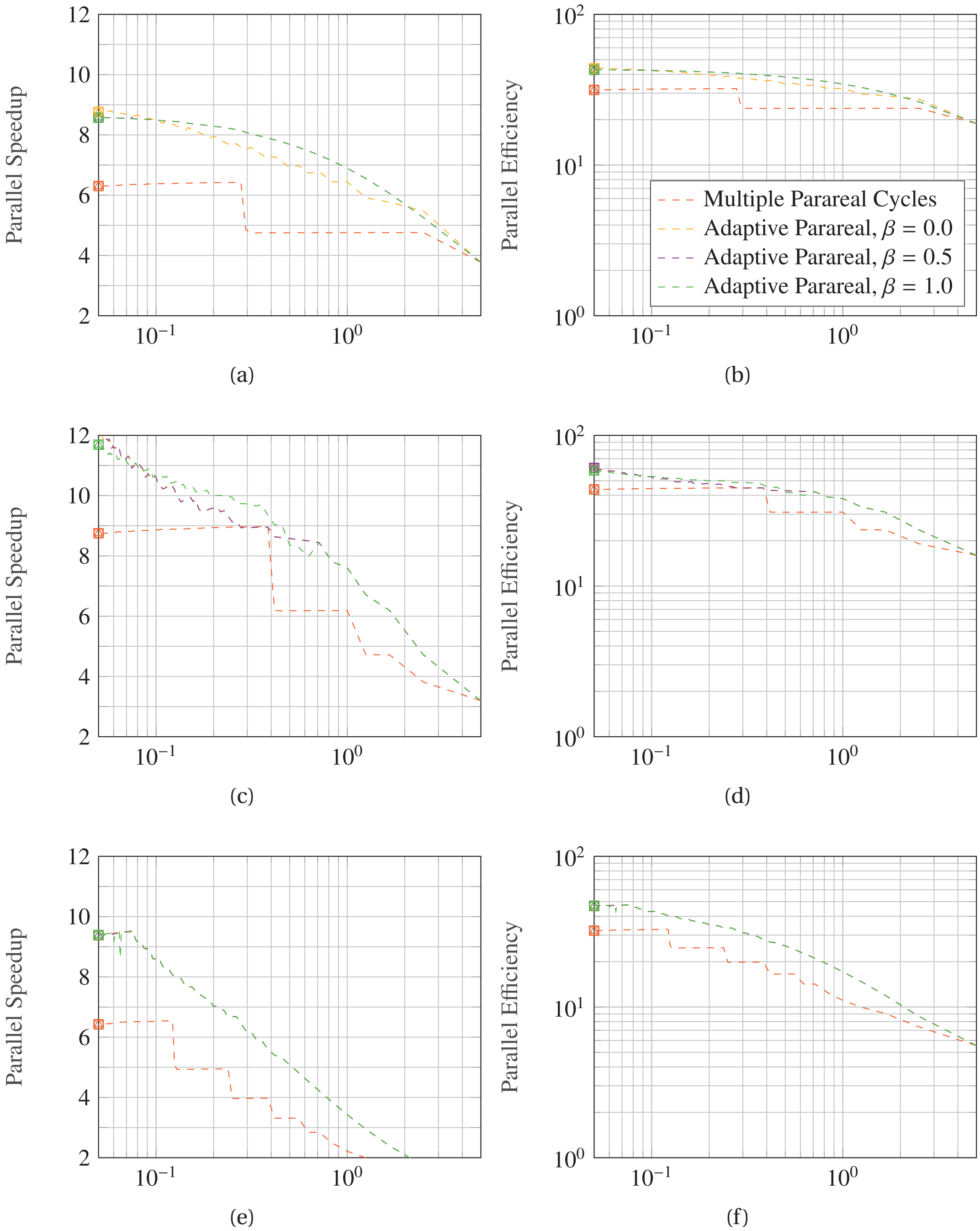
Figure C.2 Time-subdomain length for $T = 100$, $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 1ms$

Figure C.3 Number of cycles N_c for $T = 100$, $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 10ms$

Figure C.4 Time-subdomain length for $T = 100$, $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 10ms$

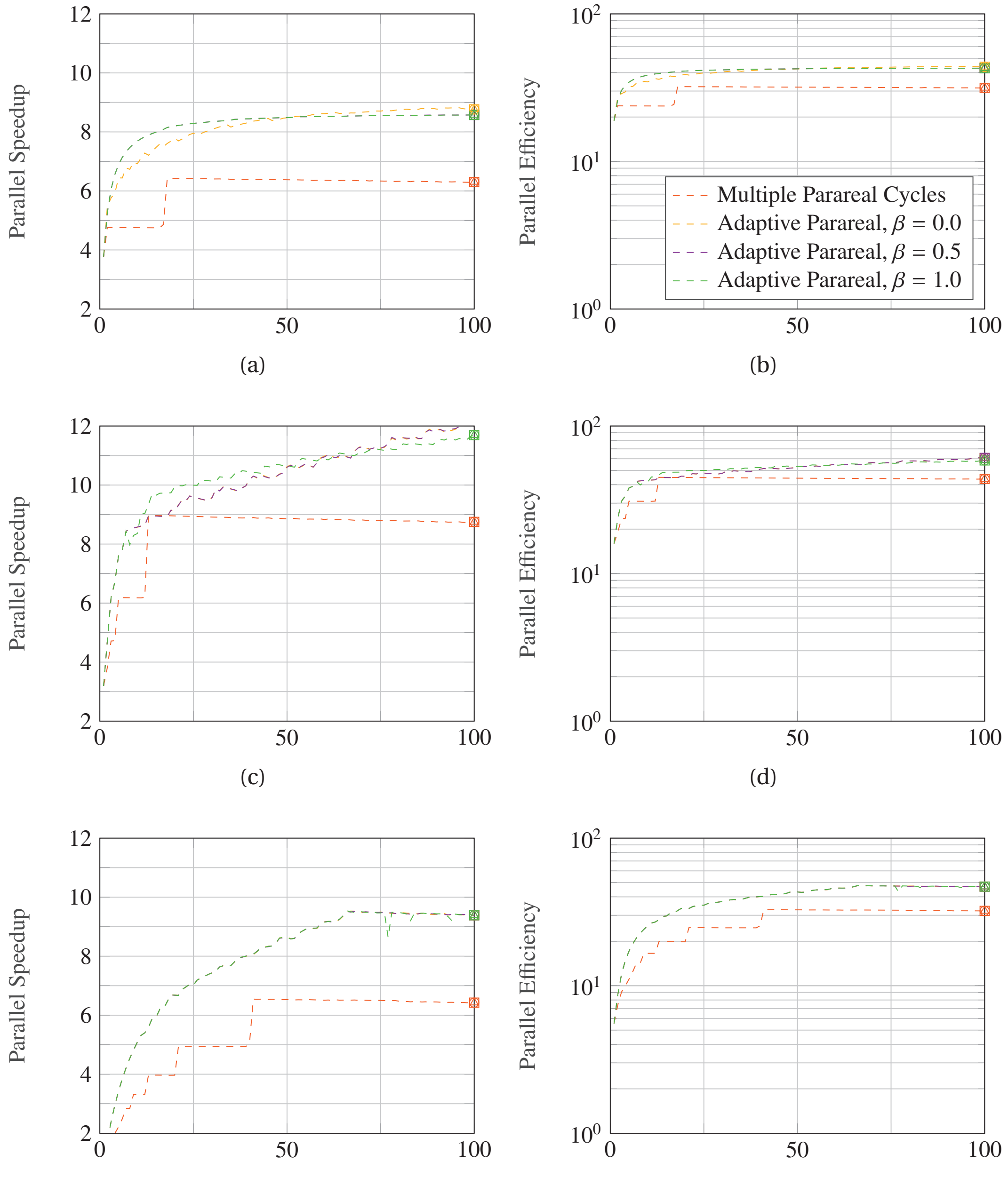
Figure C.5 Number of cycles N_c for $T = 100$, $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 100ms$

Figure C.6 Time-subdomain length for $T = 100$, $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 100ms$



(a) (b) (c) (d) (e) (f)

Figure C.1 – Parallel speedup as a function of time-subdomain length δt when integrating equation 7.30 to $T = 100$ using multiple consecutive cycles of a standard fully distributed parareal, and when using CAAP. Circles indicate a priori estimated optimal δT using equation (7.44) and equation (7.51). $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 1ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$.



(a) (b) (c) (d) (e) (f)

Figure C.2 – Parallel efficiency as a function of number of cycles n_c used in the time-parallel integration to $T = 100$. The black line represents data measured using multiple consecutive cycles of a standard fully distributed parareal, and the yellow/purple dashed lines are measured using CAAP. Circles indicate a priori estimated optimal δT using equation (7.44) and equation (7.51). $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 1ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$.

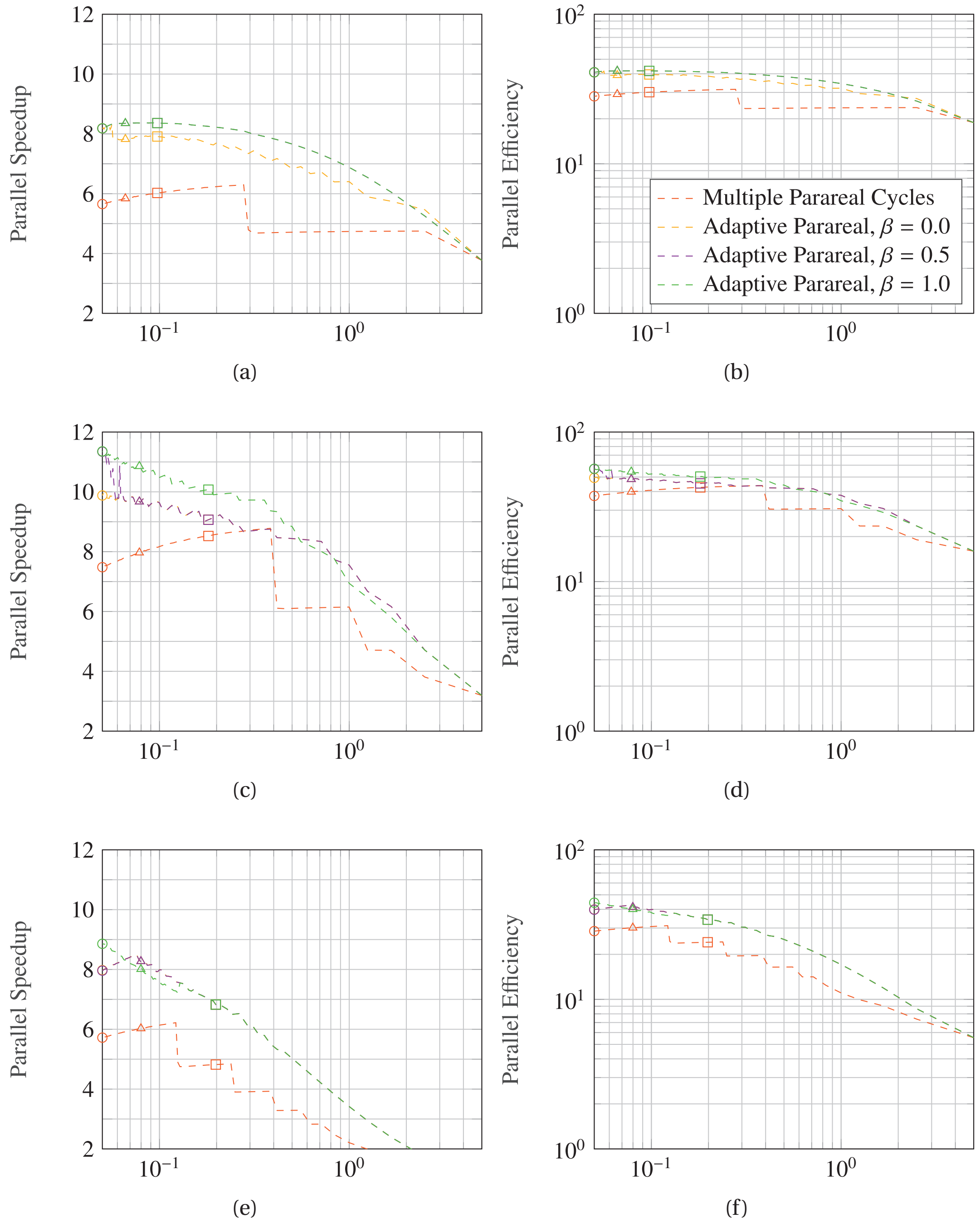
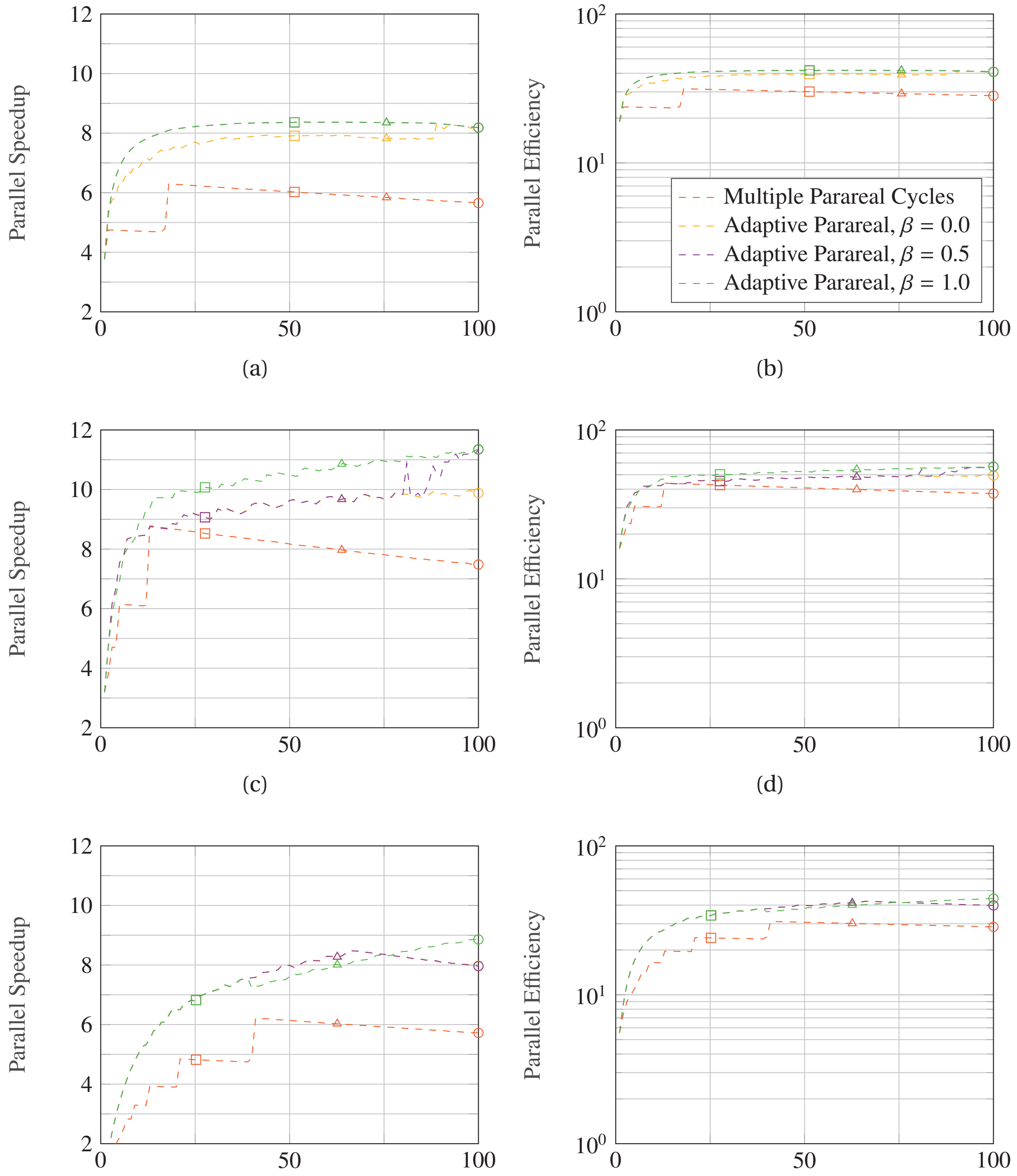


Figure C.3 – Parallel speedup as a function of timesubdomain length δt when integrating equation 7.30 to $T = 100$ using multiple consecutive cycles of a standard fully distributed parareal scheduler, and when using CAAP. Circles indicate a priori estimated optimal δT using equation 7.44 and 7.51. $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 10ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$.

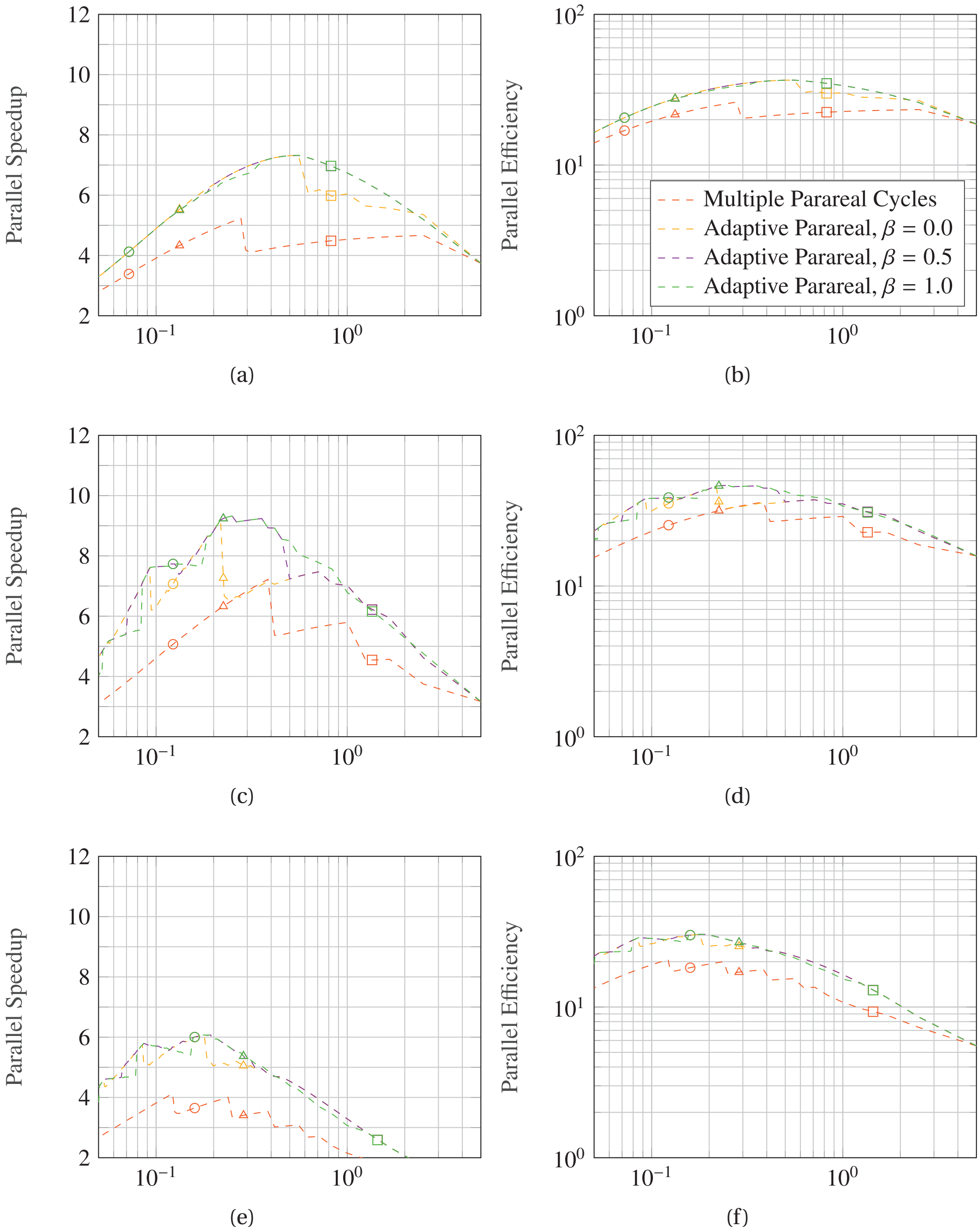


(a) (b)

(c) (d)

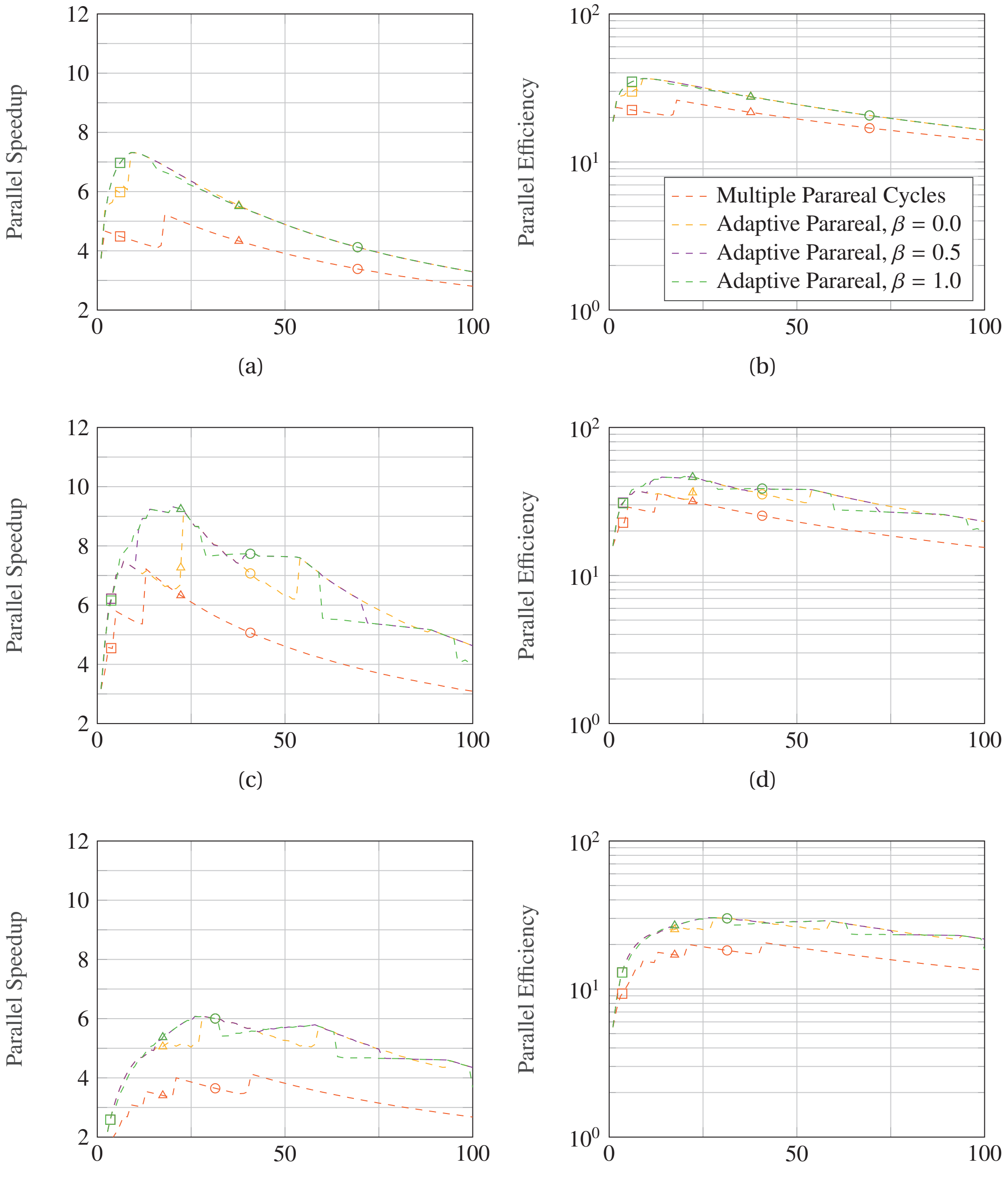
(e) (f)

Figure C.4 – Parallel speedup as a function of number of cycles n_c used in the time-parallel integration to $T = 100$. The black line represents data measured using multiple consecutive cycles of a standard fully distributed parareal, and the yellow/purple dashed lines CAAP. Circles indicate a priori estimated optimal δT using equation 7.44 and 7.51. $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 10ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$.



(a) (b) (c) (d) (e) (f)

Figure C.5 – Parallel speedup as a function of timesubdomain length δt when integrating equation 7.30 to $T = 100$ using multiple consecutive cycles of a standard fully distributed parareal, and when using CAAP. Circles indicate a priori estimated optimal δT using equation 7.44 and 7.51. $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 100ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$.



(a) (b)

(c) (d)

(e) (f)

Figure C.6 – Parallel speedup as a function of number of cycles n_c used in the time-parallel integration to $T = 100$. The black line represents data measured using multiple consecutive cycles of a standard fully distributed parareal, and the yellow/purple dashed lines are measured using CAAP. Circles indicate a priori estimated optimal δT using equation 7.44 and 7.51. $N_p = 20$, $\epsilon = 10^{-2}$, $T_c^w = 100ms$, $C_{\mathcal{F}}^w = 10000ms$. (a)-(b) $dT = 10^{-4}$, $C_{\mathcal{G}}^w = 1000ms$. (b)-(c) $dT = 10^{-3}$, $C_{\mathcal{G}}^w = 100ms$. (d)-(e) $dT = 10^{-2}$, $C_{\mathcal{G}}^w = 10ms$.

D CAAP Scheduler on 2D SHW

Appendix D contains schematic drawings of the schedulers presented in chapter 7. Unlike appendix A where the figures are for the method applied to solve the test ODE 7.30, here figures are presented of the methods applied to the 2D shallow water wave equation with $\mathcal{F}$ and $\mathcal{G}$ as outlined in section 7.2. The figures are created using time-stamp dumps from actual cluster tests. In all tests, 1 node was used in space and the integration performed until $T = 100$min and $\epsilon = 10^{-5}$ as the convergence criteria.

Figure D.1 Standard Parareal, i.e. a single cycle $n_c = 1$

Figure D.2 Multiple consecutive Parareal cycles, $n_c = 10$

Figure D.3 Adaptive Parareal, $\beta = 0.0$, $n_c = 10$

Figure D.4 Adaptive Parareal, $\beta = 0.4$, $n_c = 10$

Figure D.5 Adaptive Parareal, $\beta = 0.4$, $n_c = 10$

In all figures the green squares indicate the mpi send(s) being posted for sending the solution state to the next time-subdomain, red circles indicates a complementing receive being posted, green x indicate completion of send operation. The green arrows indicate signal flag send and receive. The figures differ from those in Appendix A in that load balancing issues due to adaptive timesteps of the exposed. In the example shown, $\beta = 0.0$ and $\beta = 0.4$ is faster than $\beta = 0.8$ because the cost of extra iterations is outways the gain from having less idle nodes.

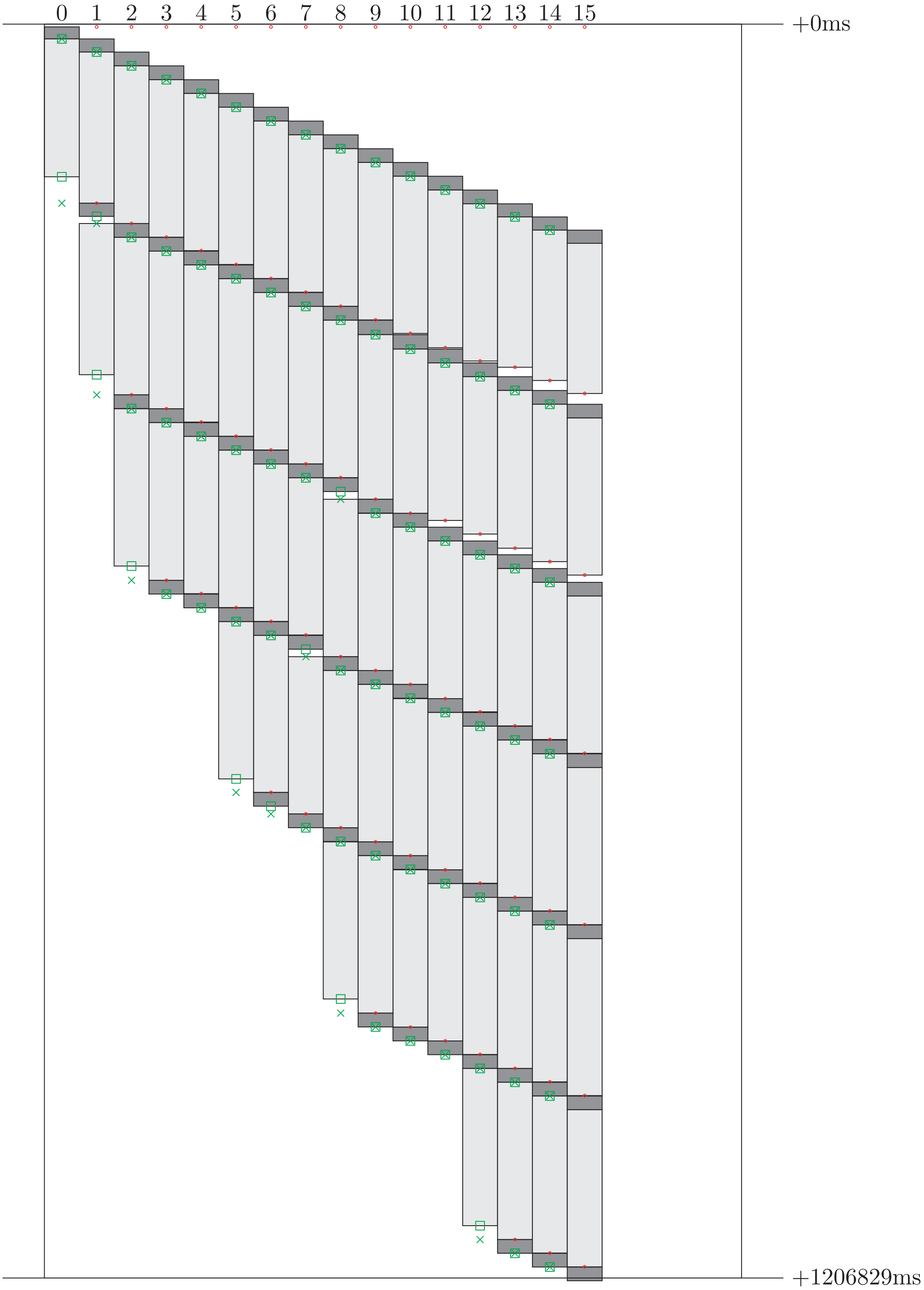


Figure D.1 – Fully distributed Parareal applied to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 1$. Integration to $T = 100$min, 1 node in space.

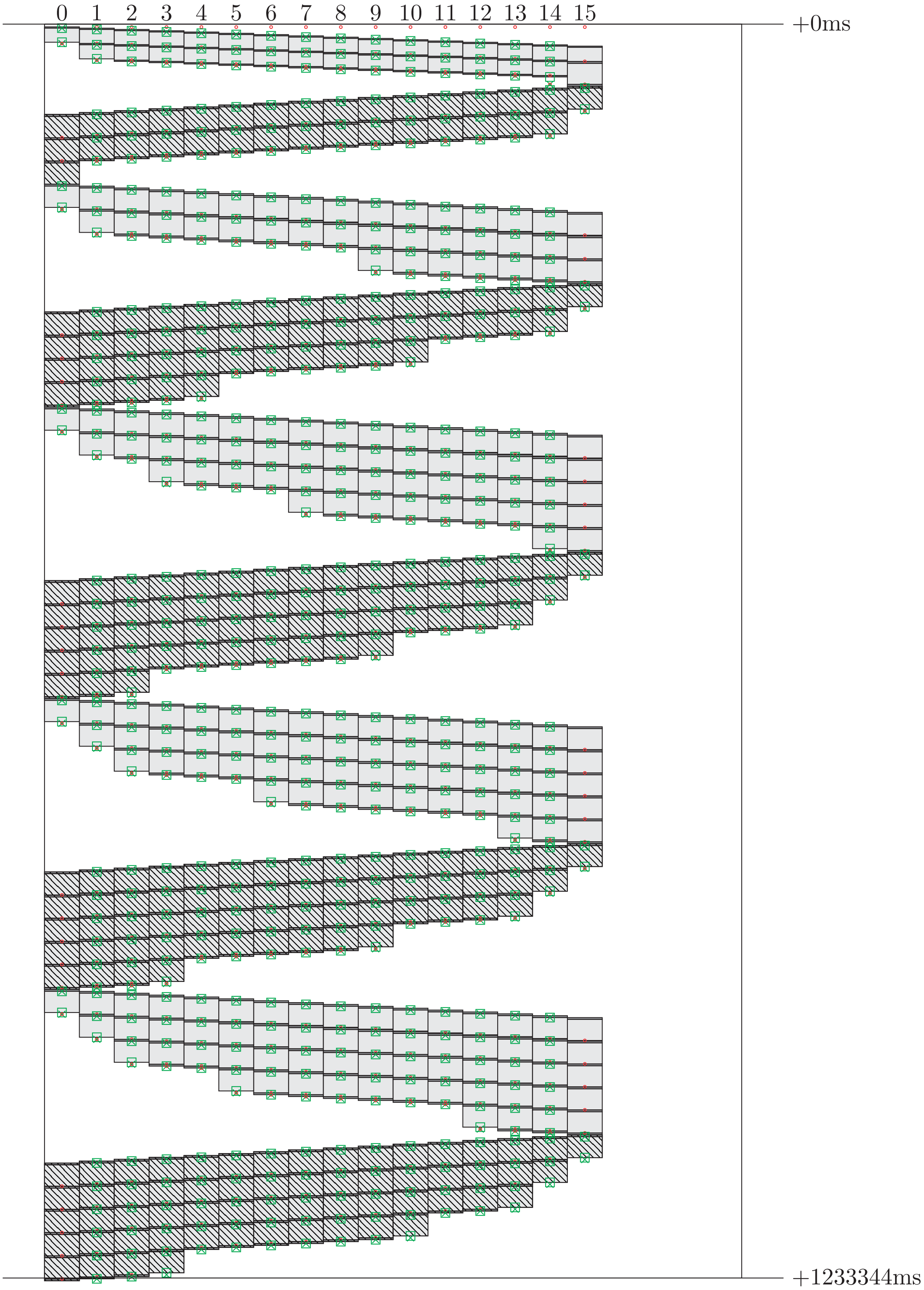


Figure D.2 – Multiple consecutive executions of the standard “fully distributed” Parareal to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 10$. Integration to $T = 100\text{min}$, 1 node in space.

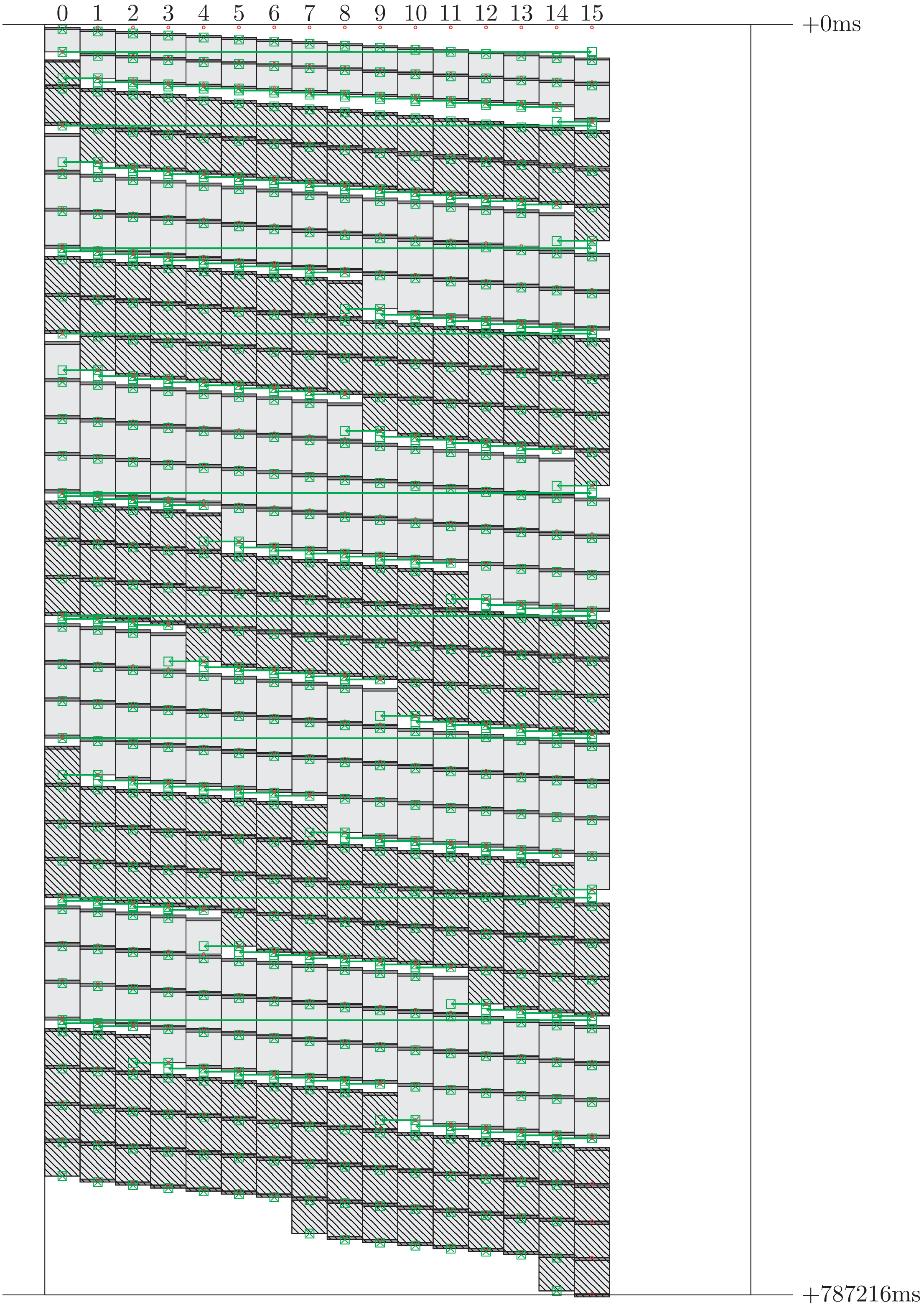


Figure D.3 – Adaptive Parareal with $\beta = 0.0$ used to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 10$. Integration to $T = 100$min, 1 node in space.

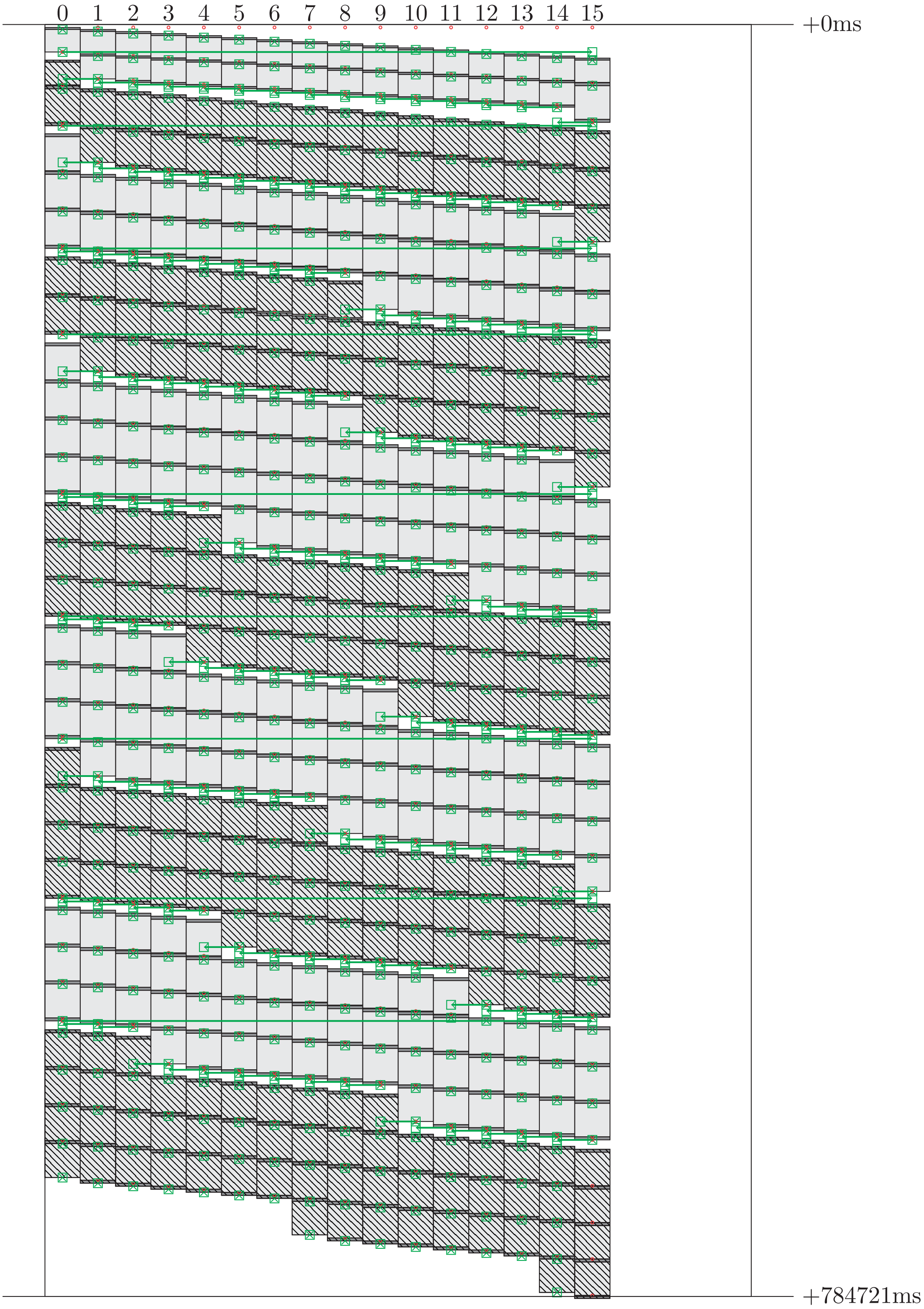


Figure D.4 – Adaptive Parareal with $\beta = 0.4$ used to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 10$. Integration to $T = 100\text{min}$, 1 node in space.

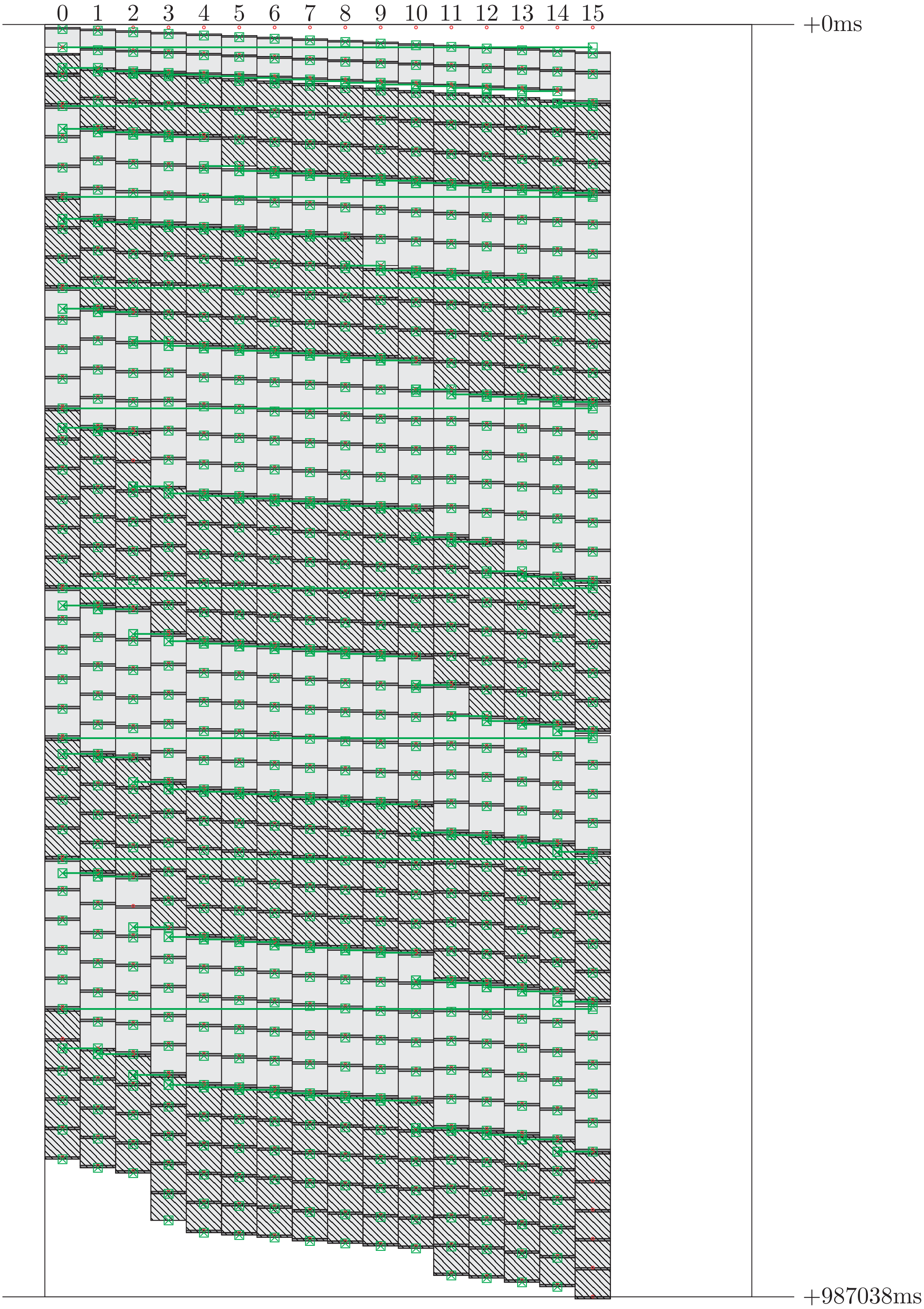


Figure D.5 – Adaptive Parareal with $\beta = 0.8$ used to solve the 2D shallow water wave equation test case using the WENO SSP-RK scheme as outlined in chapter 7. $n_c = 10$. Integration to $T = 100$min, 1 node in space.

E Llama Library Example Code

This appendix contains code examples for Llama library usage. Listing E.1 contains an unprotected code example solving the 2D euler equation in parallel using MPI. All simulation code is encapsulated within the EulerSim object for readability. The code consists only of an initialization phase and a loop. Within the loop, one time-step is computed in each round. In Listing E.2, the same code has been protected using the Llama library. In the example given, two layers are used to protect the application. A to-disk checkpoint that updates once every 25th step, and a cheaper in-memory checksum checkpoint that updates once every 5th step. Several errors are injected, and survival is demonstrated. The Llama guard choses which checkpoint to recover from depending on the failure pattern so that the loop completes correctly despite the forced termination of several ranks by raising SIGKILL.

Unprotected Code Example

Listing E.1 – Unprotected example code used to demonstrate usage of the Llama library

```
#include "eulersim.h"

void PrintPerformance(const EulerSim &, double, double);

int main(int argc, char **argv) {

    // Solving the 2D euler equation on a test-case in Parallel using MPI.
    // For implementation details, see eulersim.h and eulersim.cc
    //
    // Run with mpirun --mca io romio314 --oversubscribe -np 8 euler_example

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Make a duplicate of MPI_COMM_WORLD to use in application
    MPI_Comm my_comm;
    MPI_Comm_dup(MPI_COMM_WORLD, &my_comm);

    // Create a simulation object on a 1024x1024 grid with initial condition 6
    auto *my_sim = new EulerSim(&my_comm, 1024, 6);

    // For loop to simulate over time
    double start = MPI_Wtime();
    for (int idx = 0; idx < 100; idx++) {

        // Do a timestep my_sim.dt
        my_sim->DoTimeStep();
        my_sim->PrintStatus();

    }
    double finish = MPI_Wtime();

    // Check accuracy with respect to solution state on disk
    my_sim->SaveStateToFile();

    // Communicate completion
    PrintPerformance(*my_sim, finish, start);

    // Finalize the MPI environment.
    MPI_Finalize();

    return 0;

}

void PrintPerformance(const EulerSim &MySim, double finish, double start) {
    if (MySim.world_rank == 0) {
        std::cout << "Time to compute: " << finish - start << " seconds" << std::endl;
        std::cout << "Avg. performance: "
                  << 95.0 * (double)(MySim.cols) * (double)(MySim.cols) *
                     (double)(MySim.time_step_count) /
                     (1.0e9 * (finish - start)) << " gflops" << std::endl;
    }
}
```

Code Example protected using two-layered Llama

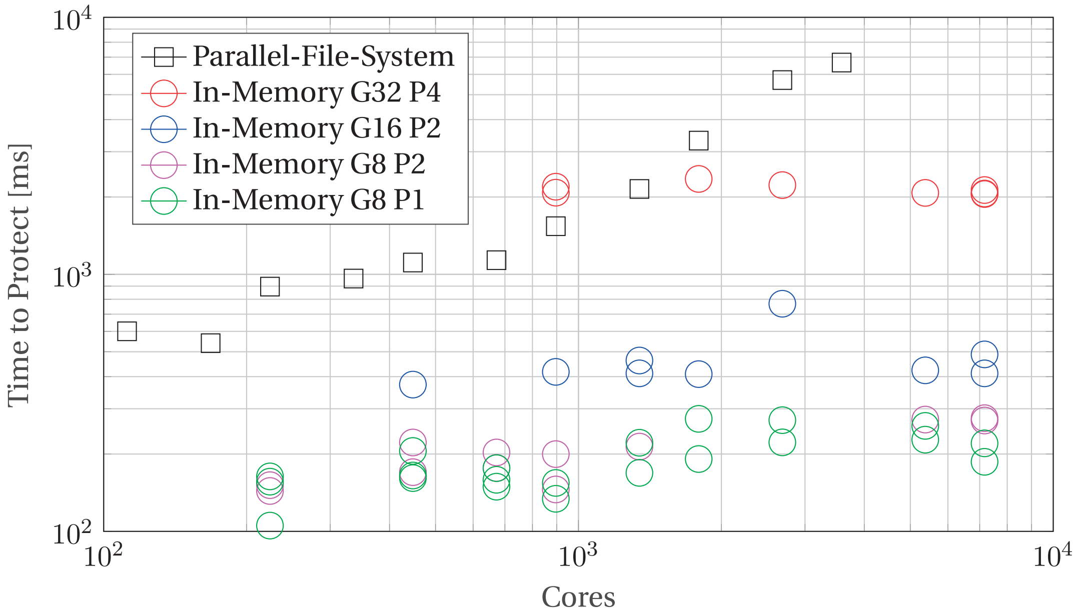
Listing E.2 – Llama usage example using two separate checkpoint levels.

```cpp
#include "eulersim.h"
#include "llama.h"

#define STEPS_PER_MEM_CHECKPOINT 5
#define STEPS_PER_DISK_CHECKPOINT 25

void PrintPerformance(const EulerSim &, double, double);

int main(int argc, char **argv) {

    // Solving the 2D euler equation - With fault-tolerance!
    //
    // Two-layer protection using an in-memory checkpoint and a disk checkpoint
    //      One checkpoint with a group-size of 4 and 2 parity codes per group
    //      One regular to-disk checkpoint
    // In this test two failures are injected:
    //      One at idx 57 losing world rank 1, 2 and 3
    //      One at idx 92 losing world rank 0
    // At the first failure, if the behavior is correct the guard is supposed to
    // recognize that the cheap checkpoint at idx 55 is insufficient for recovery
    // due to 3 simultaneous failures. Instead the guard should organize a rollback
    // to the most recent to-disk checkpoint update that was made at idx 50. An
    // additional fault is injected at idx 92, here the most recent update of the
    // in-memory checksum checkpoint can recover the lost data, the guard should
    // recognize this and execute a rollback to idx 90 by instructing the checkpoint
    // to decode the lost data from the parity codes stored.

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Make a duplicate of MPI_COMM_WORLD to use in application
    MPI_Comm my_comm;
    MPI_Comm_dup(MPI_COMM_WORLD, &my_comm);

    // Here we create our llama guard to protect our code, using 4 spare ranks
    llama::guard my_llama(my_comm, 4, true);

    // Hello world!
    my_llama.Disp("Hello world! I am worker ", my_llama.rank_worker_comm);
    MPI_Barrier(my_llama.world_comm);

    // Declare to-disk type checkpoint
    llama::checkpoint::disk my_disk_checkpoint(&my_llama);

    // Declare in-memory type checkpoint with 4 ranks and 2 parity codes per group
    llama::checkpoint::memory my_checksum_checkpoint(&my_llama, true, 1, 4, 2);

    // Initialization of simulation object
    EulerSim *my_sim = nullptr;
    if (my_llama.is_worker) {

        // Create a simulation object on a 1024x1024 distributed grid
        // using initial condition 6
        my_sim = new EulerSim(&my_comm, 1024, 6);


        // Mark all arrays that need to be protected for easy and quick repair
        my_llama.ProtectArray<double>("Density", my_sim->density,
                                            my_sim->rows, my_sim->cols);
        my_llama.ProtectArray<double>("VelocityX", my_sim->velocity_x,
                                            my_sim->rows, my_sim->cols);
        my_llama.ProtectArray<double>("VelocityY", my_sim->velocity_y,
                                            my_sim->rows, my_sim->cols);
        my_llama.ProtectArray<double>("Pressure", my_sim->pressure,
                                            my_sim->rows, my_sim->cols);
        my_llama.ProtectArray<double>("Energy", my_sim->energy,
                                            my_sim->rows, my_sim->cols);
    }

    // Do 100 time steps
    double start = MPI_Wtime();
    for (int idx = 0; idx < 100; idx++) {

        // Check status of active workers
        int llama_status = my_llama.CheckStatus(&idx);

        // Check return status of repair is needed
        if (llama_status == LLAMA_STATUS_REPAIR) {

            // Communicate status
            my_llama.Disp("Detected failure - performing rollback to idx:", idx);
            MPI_Barrier(my_comm);

            // Clean-up old data structures
            delete my_sim;

            // Regenerate data structures
            my_sim = new EulerSim(&my_comm, 1024, 6);

            // Recover data from most recent checkpoint
            my_llama.RecoverArray<double>(idx, "Density", my_sim->density,
                                                my_sim->rows, my_sim->cols);
            my_llama.RecoverArray<double>(idx, "VelocityX", my_sim->velocity_x,
                                                my_sim->rows, my_sim->cols);
            my_llama.RecoverArray<double>(idx, "VelocityY", my_sim->velocity_y,
                                                my_sim->rows, my_sim->cols);
            my_llama.RecoverArray<double>(idx, "Pressure", my_sim->pressure,
                                                my_sim->rows, my_sim->cols);
            my_llama.RecoverArray<double>(idx, "Energy", my_sim->energy,
                                                my_sim->rows, my_sim->cols);

            // Update counter in simulation object
            my_sim->time_step_count = idx;

        }

        // Update address of arrays to protect
        my_llama.SetArrayAddress<double>("Density", my_sim->density);
        my_llama.SetArrayAddress<double>("VelocityX", my_sim->velocity_x);
        my_llama.SetArrayAddress<double>("VelocityY", my_sim->velocity_y);
        my_llama.SetArrayAddress<double>("Pressure", my_sim->pressure);
        my_llama.SetArrayAddress<double>("Energy", my_sim->energy);




        // Update checkpoint if need be
        if (idx % STEPS_PER_DISK_CHECKPOINT == 0) {
            my_llama.Disp("Updating disk checkpoint at idx:", idx);
            my_disk_checkpoint.Update(idx);
        }
        if (idx % STEPS_PER_MEM_CHECKPOINT == 0) {
            my_llama.Disp("Updating in-memory checksum checkpoint at idx:", idx);
            my_checksum_checkpoint.Update(idx);
        }

        // Do some actual work (a timestep)
        my_llama.Disp("Computing at idx:", idx);
        my_sim->DoTimeStep();

        // Kill one or more ranks
        if (idx == 57 && 0 < my_llama.rank_world_comm &&
                                my_llama.rank_world_comm < 4 ) {
            my_llama.Disp("Raising sigkill at idx:", idx);
            raise(SIGKILL);
        }
        // Kill one or more ranks
        if (idx == 92 && (my_llama.rank_world_comm == 0)) {
            my_llama.Disp("Raising sigkill at idx:", idx);
            raise(SIGKILL);
        }

    }
    double finish = MPI_Wtime();

    // Check accuracy with respect to reference solution
    my_sim->CheckAccuracy();
    my_sim->SaveStateToFile();

    // Print compute time and performance
    PrintPerformance(*my_sim, finish, start);

    // Finalize the Llama protection environment.
    my_llama.Finalize();

    // Finalize the MPI environment.
    MPI_Finalize();

    return 0;

}

void PrintPerformance(const EulerSim &MySim, double finish, double start) {
    if (MySim.world_rank == 0) {
        std::cout << "Time to compute: " << finish - start << " seconds" << std::endl;
        std::cout << "Avg. performance: "
                  << 95.0 * (double)(MySim.cols) * (double)(MySim.cols) *
                     (double)(MySim.time_step_count) /
                     (1.0e9 * (finish - start)) << " gflops" << std::endl;
    }
}
```

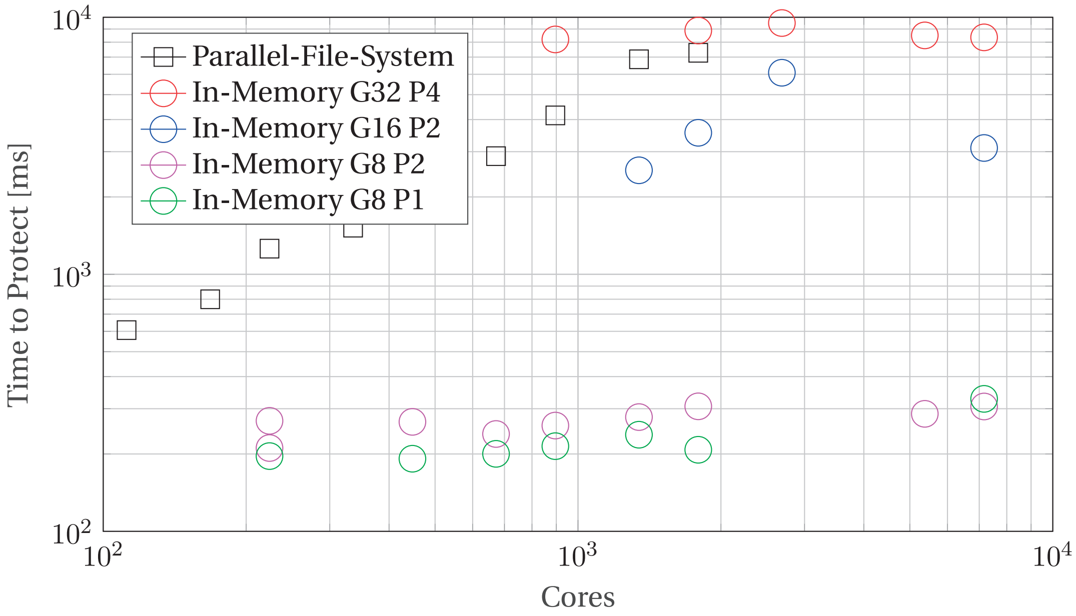
F Llama Library Scaling Tests

Appendix F contains weak and strong scaling measurements of the time to update array protection in a checkpoint and of the time to recover lost data from a checkpoint. The timings were used to compute the speed of data protection/recovery presented in section 11.3 of chapter 11. Scaling tests for the cost of updating a checkpoint are given in figure F.1 and F.2. Scaling tests for the cost of updating a checkpoint are given in figures F.3 and F.4.

Updating Checkpoint Protection - Weak Scaling



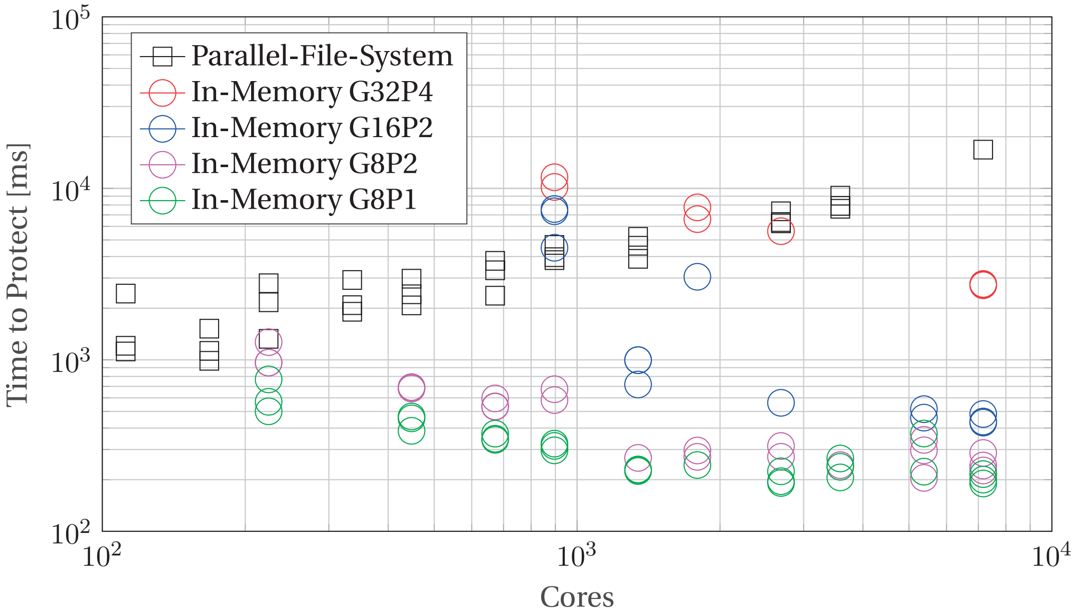
(a) Weak Scaling – 1.5MB per Array per Core



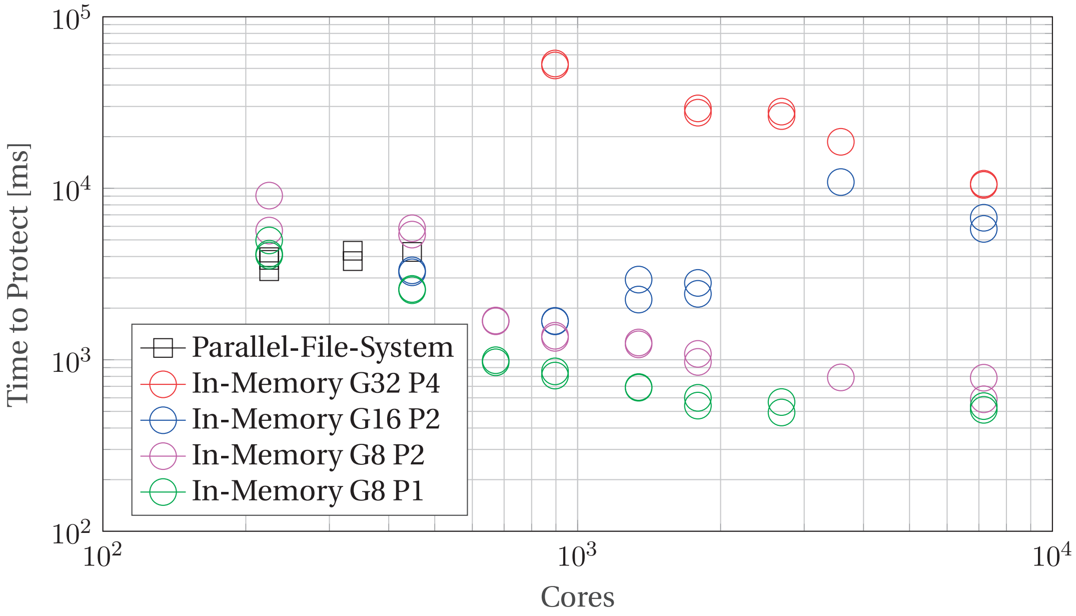
(b) Weak Scaling – 12MB per Array per Core

Figure F.1 – Weak scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints.

Updating Checkpoint Protection - Strong Scaling



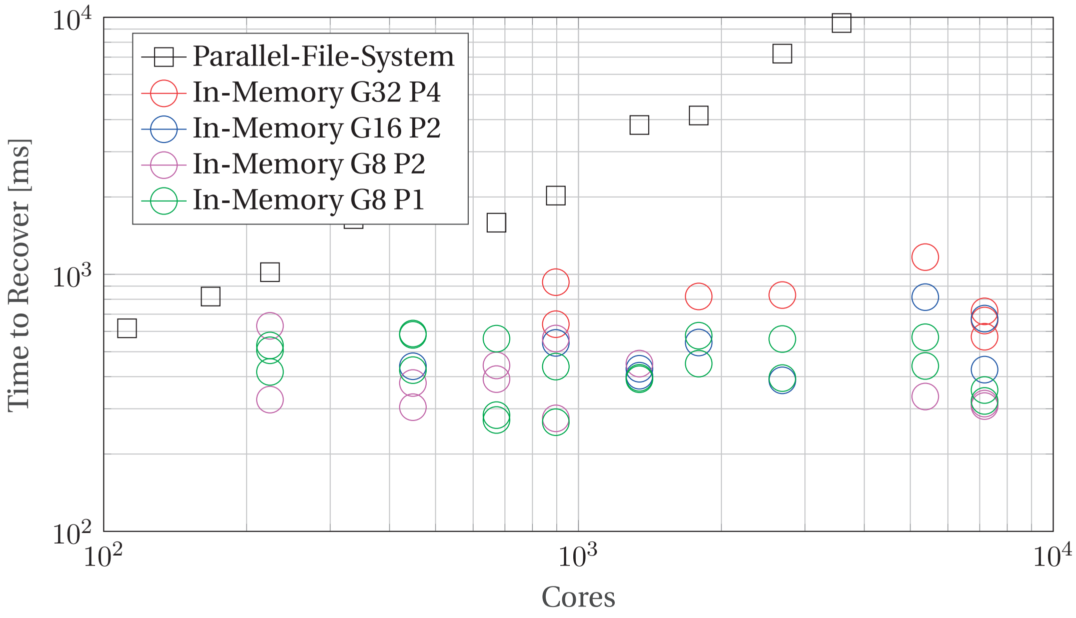
(a) Strong Scaling – 16GB Distributed Array



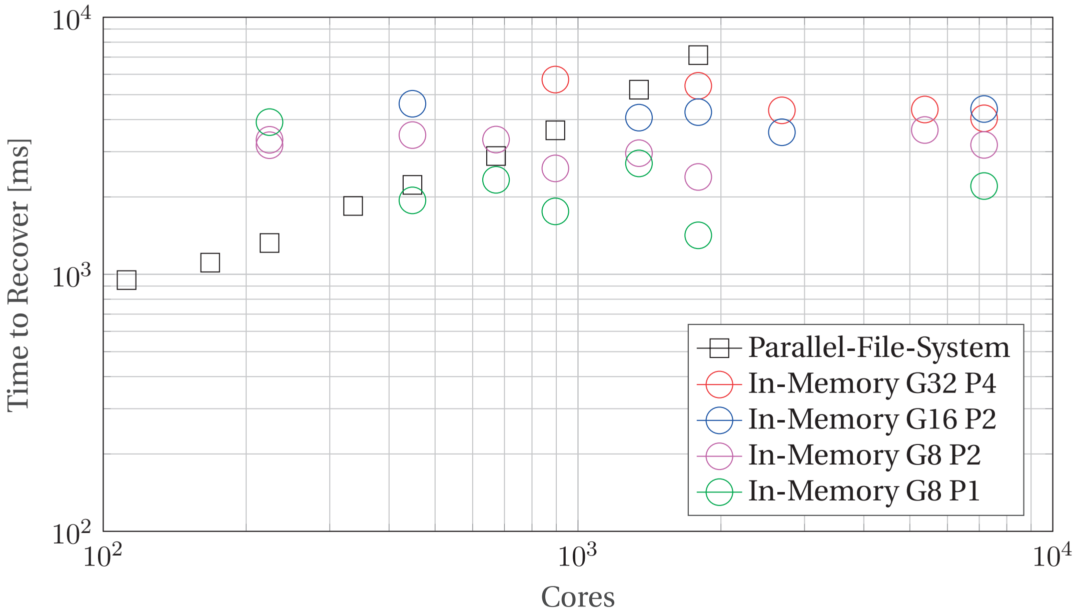
(b) Strong Scaling – 128GB Distributed Array

Figure F.2 – Strong scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding 4 parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system.

Data Recovery - Weak Scaling



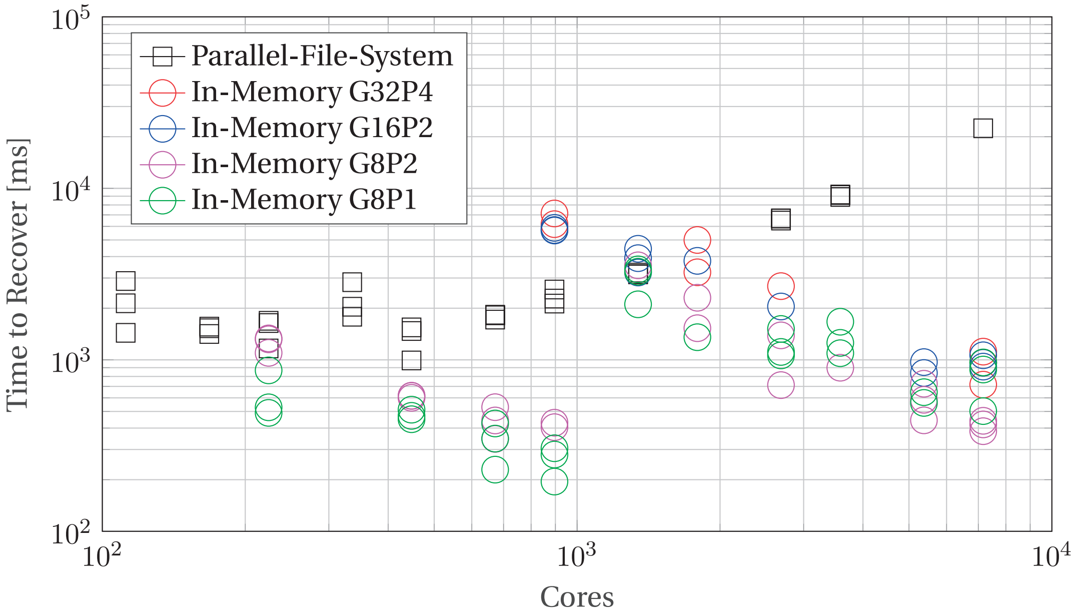
(a) Weak Scaling – 1.5MB per Array per Core



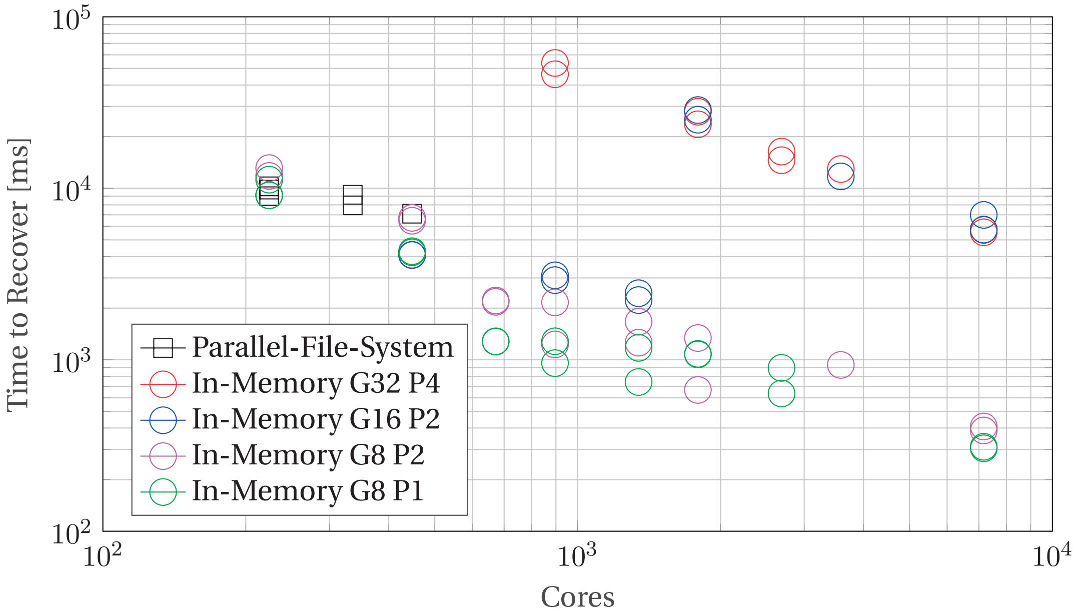
(b) Weak Scaling – 12MB per Array per Core

Figure F.3 – Weak scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints.

Data Recovery - Strong Scaling



(a) Strong Scaling – 16GB Distributed Array



(b) Strong Scaling – 128GB Distributed Array

Figure F.4 – Strong scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding 4 parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system.

Bibliography

[1] A. Aggarwal, R. J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 297–308. ACM, 1989.

[2] S. Alowayyed, D. Groen, P. V. Coveney, and A. G. Hoekstra. Multiscale computing in the exascale era. *Journal of Computational Science*, 22:15 – 25, 2017.

[3] G. Alvarez, M. S. Summers, D. E. Maxwell, M. Eisenbach, J. S. Meredith, J. M. Larkin, J. Levesque, T. A. Maier, P. R. C. Kent, E. F. D'Azevedo, and T. C. Schulthess. New algorithm to enable 400+ tflop/s sustained performance in simulations of disorder effects in high-tc superconductors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 61:1–61:10, Piscataway, NJ, USA, 2008. IEEE Press.

[4] C. Amante. Etopo1 1 arc-minute global relief model: procedures, data sources and analysis. *http://www. ngdc. noaa. gov/mgg/global/global. html*, 2009.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[6] Argonne Leadership Computing Facility (ALCF). Aurora 2021 Early Science Program: Data and Learning Call For Proposals, 2018. link [Acccessed: 06.06.2018].

[7] R. A. Ashraf, S. Hukerikar, and C. Engelmann. Shrink or substitute: Handling process failures in hpc systems using in-situ recovery. *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 178–185, 2018.

[8] Association for Computing Machinery (ACM). Gordon Bell Prize, 2018. link [Acccessed: 04.06.2018].

[9] E. Aubanel. Scheduling of tasks in the parareal algorithm. *Parallel Comput.*, 37(3):172–182, Mar. 2011.

[10] G. Aupy, A. Benoit, T. Hérault, Y. Robert, and J. Dongarra. Optimal checkpointing period: Time vs. energy. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 203–214. Springer, 2013.

[11] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni. Checkpointing algorithms and fault prediction. *Journal of Parallel and Distributed Computing*, 74(2):2048–2064, 2014.

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[13] N. Bajunaid and D. A. Menasce. Analytic models of checkpointing for concurrent component-based software systems. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 245–256, New York, NY, USA, 2017. ACM.

[14] G. Bal. On the convergence and the stability of the parareal algorithm to solve partial differential equations. In *Domain decomposition methods in science and engineering*, pages 425–432. Springer, 2005.

[15] P. Balaprakash, L. A. B. Gomez, M.-S. Bouguerra, S. M. Wild, F. Cappello, and P. D. Hovland. Analysis of the tradeoffs between energy and run time for multilevel checkpointing. In S. A. Jarvis, S. A. Wright, and S. D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 249–263, Cham, 2015. Springer International Publishing.

[16] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: The architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, Nov 2008.

[17] J. Barr. Exascale Challenges, Scientific Computing World, 2014. link [Acccessed: 28.07.2018].

[18] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994.

[19] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. Which verification for soft error detection? In *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*, pages 2–11. IEEE, 2015.

[20] L. Bautista Gomez and F. Cappello. Detecting silent data corruption through data dynamic monitoring for scientific applications. In *ACM SIGPLAN Notices*, volume 49, pages 381–382. ACM, 2014.

[21] L. Bautista-Gomez and F. Cappello. Detecting silent data corruption for extreme-scale mpi applications. In *Proceedings of the 22Nd European MPI Users' Group Meeting*, EuroMPI '15, pages 12:1–12:10, New York, NY, USA, 2015. ACM.

[22] L. Bautista-Gomez, T. Ropars, N. Maruyama, F. Cappello, and S. Matsuoka. Hierarchical clustering strategies for fault tolerance in large scale hpc systems. In *IEEE Cluster 2012*, 2012.

[23] L. Bautista-Gomez and K. Sierocinski. FTI Repository, 2018. link [Acccessed: 28.07.2018].

[24] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[25] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith. Unprotected computing: A large-scale study of dram raw error rate on a supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 55:1–55:11, Piscataway, NJ, USA, 2016. IEEE Press.

[26] J. Bédorf, E. Gaburov, M. S. Fujii, K. Nitadori, T. Ishiyama, and S. P. Zwart. 24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 54–65, Piscataway, NJ, USA, 2014. IEEE Press.

[27] G. Bell, D. H. Bailey, J. Dongarra, A. H. Karp, and K. Walsh. A look back on 30 years of the gordon bell prize. *Int. J. High Perform. Comput. Appl.*, 31(6):469–484, Nov. 2017.

[28] A. Benoit, A. Cavelan, F. Cappello, P. Raghavan, Y. Robert, and H. Sun. Identifying the right replication level to detect and correct silent errors at scale. In *Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale*, pages 31–38. ACM, 2017.

[29] A. Benoit, A. Cavelan, V. Le Fèvre, and Y. Robert. Optimal checkpointing period with replicated execution on heterogeneous platforms. In *Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale*, FTXS '17, pages 9–16, New York, NY, USA, 2017. ACM.

[30] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. Towards optimal multi-level checkpointing. *IEEE Transactions on Computers*, 66(7):1212–1226, 2017.

[31] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications*, 29(4):403–421, 2015.

[32] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. 2008.

[33] M. Bernaschi, M. Bisson, T. Endo, S. Matsuoka, M. Fatica, and S. Melchionna. Petaflop biofluidics simulations on a two million-core system. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 4:1–4:12, New York, NY, USA, 2011. ACM.

[34] M. Bernaschi, M. Bisson, M. Fatica, and S. Melchionna. 20 petaflops simulation of proteins suspensions in crowding conditions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 2:1–2:11, New York, NY, USA, 2013. ACM.

[35] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 275–278, New York, NY, USA, 2015. ACM.

[36] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. Exploring partial replication to improve lightweight silent data corruption detection for hpc applications. In *European Conference on Parallel Processing*, pages 419–430. Springer, 2016.

[37] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. Toward general software level silent data corruption detection for parallel applications. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3642–3655, 2017.

[38] L. A. Berry, W. Elwasif, J. M. Reynolds-Barredo, D. Samaddar, R. Sanchez, and D. E. Newman. Event-based parareal: A data-flow based implementation of parareal. *Journal of Computational Physics*, 231(17):5945–5954, 2012.

[39] S. Biswas, B. R. d. Supinski, M. Schulz, D. Franklin, T. Sherwood, and F. T. Chong. Exploiting data similarity to reduce memory footprints. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 152–163, May 2011.

[40] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of user-level failure mitigation support in mpi. *Computing*, 95(12):1171–1184, 2013.

[41] J. Bodart, S. Gratton, X. Vasseur, and T. Lunet. Stable time-parallel integration of advection dominated problems using parareal with space coarsening. In *7th Workshop on Parallel-in-Time, 2 May - 5 May 2018 (Roscoff, France)*. PINT.

[42] G. Bosilca and A. Bouteiller. ULFM MPI Repository, 2018. link [Acccessed: 28.07.2018].

[43] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Using group replication for resilience on exascale systems. *The International Journal of High Performance Computing Applications*, 28(2):210–224, 2014.

[44] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13, 2013. Metaheuristics on GPUs.

[45] A. G. Bromley. Charles babbage's analytical engine, 1838. *IEEE Annals of the History of Computing*, 20(4):29–45, 1998.

[46] P. Broughton. The first predicted return of comet halley. *Journal for the History of Astronomy*, 16:123, 1985.

[47] J. Calhoun, M. Snir, L. N. Olson, and W. D. Gropp. Towards a more complete understanding of sdc propagation. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 131–142, New York, NY, USA, 2017. ACM.

[48] C. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. D. Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. Kirby, and S. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205 – 219, 2015.

[49] C. D. Cantwell and A. S. Nielsen. A minimally intrusive low-memory approach to resilience for existing transient solvers. *Journal of Scientific Computing*, Jul 2018.

[50] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman. System-level scalable checkpoint-restart for petascale computing. In *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on*, pages 932–941. IEEE, 2016.

[51] F. Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[52] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[53] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations*, 1(1), 2014.

[54] F. Cappello, R. Gupta, S. Di, E. Constantinescu, T. Peterka, and S. M. Wild. Understanding and improving the trust in results of numerical simulations and scientific data analytics. In D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer, editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 545–556, Cham, 2018. Springer International Publishing.

[55] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. On the impact of process replication on executions of large-scale parallel applications with coordinated checkpointing. *Future Generation Computer Systems*, 51:7–19, 2015.

[56] A. Cavelan, A. Fang, A. A. Chien, and Y. Robert. Resilient n-body tree computations with algorithm-based focused recovery: Model and performance analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 158–178. Springer, 2017.

[57] G. Ceder and K. Persson. How supercomputers will yield a golden age of materials science. *Scientific American, Dec*, 2013.

[58] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[59] C.-S. Chang, M. Greenwald, K. Riley, K. Antypas, R. Coffey, E. Dart, S. Dosanjh, R. Gerber, J. Hack, I. Monga, et al. Fusion energy sciences exascale requirements review. an office of science review sponsored jointly by advanced scientific computing research and fusion energy sciences, january 27-29, 2016, gaithersburg, maryland. Technical report, USDOE Office of Science (SC), Washington, DC (United States). Offices of Advanced Scientific Computing Research and Fusion Energy Sciences, 2017.

[60] J. H. Chaudhry, D. Estep, S. Tavener, V. Carey, and J. Sandelin. A posteriori error analysis of two-stage computation methods with application to efficient discretization and the parareal algorithm. *SIAM Journal on Numerical Analysis*, 54(5):2974–3002, 2016.

[61] F. Chen, J. S. Hesthaven, Y. Maday, and A. S. Nielsen. An adjoint approach for stabilizing the parareal method. Technical report, 2015.

[62] F. Chen, J. S. Hesthaven, and X. Zhu. On the use of reduced basis methods to accelerate and stabilize the parareal method. In *Reduced Order Methods for modeling and computational reduction*, pages 187–214. Springer, 2014.

[63] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, 2008.

[64] Z. Chen and J. J. Dongarra. Condition numbers of gaussian random matrices. *SIAM Journal on Matrix Analysis and Applications*, 27(3):603–620, 2005.

[65] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller. Understanding soft error resiliency of bluegene/q compute chip through hardware proton irradiation and software fault injection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 587–596. IEEE Press, 2014.

[66] A. Chien, P. Balaji, P. Beckman, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, R. Schreiber, et al. Versioned distributed arrays for resilience in scientific applications: Global view resilience. *Procedia Computer Science*, 51:29–38, 2015.

[67] A. Chien, P. Balaji, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, J. Hammond, I. Laguna, et al. Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. *The International Journal of High Performance Computing Applications*, 31(6):564–590, 2017.

[68] China Daily. China's 3rd exascale supercomputer prototype set for 2018 launch, 2017. link [Acccessed: 06.06.2018].

[69] China Daily. Planned supercomputer would be 10 times faster than today's No 1, May 2018. link [Acccessed: 06.06.2018].

[70] F. Chouly and A. Lozinski. Parareal multi-model numerical zoom for parabolic multiscale problems. *Comptes Rendus Mathematique*, 352(6):535 – 540, 2014.

[71] A. J. Christlieb, C. B. Macdonald, and B. W. Ong. Parallel high-order integrators. *SIAM Journal on Scientific Computing*, 32(2):818–835, 2010.

[72] A. C. Clairaut, J. J. Lefrançois de Lalande, and N.-R. Lepaute. *Théorie du mouvement des comètes*. Michel Lambert, 1760.

[73] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak. Silent data corruption — myth or reality? In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 108–109, June 2008.

[74] I. Cores, G. Rodríguez, P. González, R. R. Osorio, et al. Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes. *New Generation Computing*, 31(3):163–185, 2013.

[75] P. H. Cowell and A. C. De la Cherois Crommelin. *Investigation of the Motion of Halley's Comet from 1759 to 1910*. Neill & Company, limited, 1910.

[76] R. Croce, D. Ruprecht, and R. Krause. Parallel-in-space-and-time simulation of the three-dimensional, unsteady navier-stokes equations for incompressible flow. In *Modeling, Simulation and Optimization of Complex Processes-HPSC 2012*, pages 13–23. Springer, 2014.

[77] X. Dai and Y. Maday. Stable parareal in time method for first-and second-order hyperbolic systems. *SIAM Journal on Scientific Computing*, 35(1):A52–A78, 2013.

[78] Z. Dai and Y. Zhang. Partition, construction, and enumeration of m–p invertible matrices over finite fields. *Finite Fields and Their Applications*, 7(3):428–440, 2001.

[79] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future generation computer systems*, 22(3):303–312, 2006.

[80] D. Dauwe, R. Jhaveri, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. Optimizing checkpoint intervals for reduced energy use in exascale systems. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, Oct 2017.

[81] D. W. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. Resilience-aware resource management for exascale computing systems. *IEEE Transactions on Sustainable Computing*, 2018.

[82] D. A. G. De Oliveira, L. L. Pilla, M. Hanzich, V. Fratin, F. Fernandes, C. Lunardi, J. M. Cela, P. O. A. Navaux, L. Carro, and P. Rech. Radiation-induced error criticality in modern hpc parallel accelerators. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 577–588. IEEE, 2017.

[83] S. Di, L. Bautista-Gomez, and F. Cappello. Optimization of a multilevel checkpoint model with uncertain execution scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 907–918, Piscataway, NJ, USA, 2014. IEEE Press.

[84] S. Di, E. Berrocal, and F. Cappello. An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 271–280. IEEE, 2015.

[85] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of multi-level checkpoint model for large scale hpc applications. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1181–1190. IEEE, 2014.

[86] V. Dolean, P. Jolivet, and F. Nataf. *An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2015.

[87] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, 8(2):6:1–6:29, June 2011.

[88] J. Dongarra, T. Herault, and Y. Robert. Fault tolerance techniques for high-performance computing. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer, 2015.

[89] J. Dongarra and M. A. Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[90] J. Dongarra, M. A. Heroux, and P. Luszczek. Hpcg benchmark: a new metric for ranking high performance computing systems. *Knoxville, Tennessee*, 2015.

[91] J. Dongarra, M. A. Heroux, and P. Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications*, 30(1):3–10, 2016.

[92] J. Dongarra, J. Hittinger, J. Bell, L. Chacon, R. Falgout, M. Heroux, P. Hovland, E. Ng, C. Webster, and S. Wild. Applied mathematics research for exascale computing. Technical Report No. LLNL-TR-651000, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Livermore, CA,, 2014.

[93] J. J. Dongarra. 30 Years of Supercomputing: History of TOP500. Slides used for invited talk at the International Symposium: New Horizons of Computational Science with Heterogeneous Many-Core Processors at Okochi Hall, Wako campus, RIKEN, Japan, February 2018.

[94] J. J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[95] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices*, 47(8):225–234, 2012.

[96] A. Edelman. Eigenvalues and condition numbers of random matrices. *SIAM Journal on Matrix Analysis and Applications*, 9(4):543–560, 1988.

[97] A. Eghbal, A. G. Gerber, and E. Aubanel. Acceleration of unsteady hydrodynamic simulations using the parareal algorithm. *Journal of Computational Science*, 2016.

[98] M. Eisenbach, C.-G. Zhou, D. M. Nicholson, G. Brown, J. Larkin, and T. C. Schulthess. A scalable method for ab initio computation of free energies in

nanoscale systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 64:1–64:8, New York, NY, USA, 2009. ACM.

[99] N. El-Sayed and B. Schroeder. Reading between the lines of failure logs: Understanding how hpc systems fail. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[100] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[101] W. R. Elwasif, S. S. Foley, D. E. Bernholdt, L. A. Berry, D. Samaddar, D. E. Newman, and R. Sanchez. A dependency-driven formulation of parareal: parallel-in-time solution of pdes as a many-task application. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, pages 15–24. ACM, 2011.

[102] C. Engelmann and S. Böhm. Redundant execution of hpc applications with mr-mpi.

[103] European Commission - Press release. Commission proposes to invest EUR 1 billion in world-class European supercomputers, 2018. link [Acccessed: 06.06.2018].

[104] European Commission - Press release. EU budget: Commission proposes EUR 9.2 billion investment in first ever digital programme, 2018. link [Acccessed: 07.06.2018].

[105] D. J. Evans and B. Sanugi. A parallel runge-kutta integration method. *Parallel computing*, 11(2):245–251, 1989.

[106] G. E. Fagg and J. J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 346–353. Springer, 2000.

[107] R. D. Falgout, S. Friedhoff, T. V. Kolev, S. P. MacLachlan, and J. B. Schroder. Parallel time integration with multigrid. *SIAM Journal on Scientific Computing*, 36(6):C635–C661, 2014.

[108] C. Farhat and M. Chandesris. Time-decomposed parallel time-integrators: theory and feasibility studies for fluid, structure, and fluid–structure applications. *International Journal for Numerical Methods in Engineering*, 58(9):1397–1434, 2003.

[109] C. Farhat, J. Cortial, C. Dastillung, and H. Bavestrello. Time-parallel implicit integrators for the near-real-time prediction of linear structural dynamic responses. *International journal for numerical methods in engineering*, 67(5):697–724, 2006.

[110] M. Feldman. Thomas Sterling Talks Exascale, Chinese HPC, Machine Learning, and Non-von Neumann Architectures, 2018. link [Acccessed: 30.07.2018].

[111] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 385–396. ACM, 2010.

[112] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, and R. Brightwell. rmpi: increasing fault resiliency in a message-passing environment.

[113] K. Ferreira, J. Stearley, J. H. Laros III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 44. ACM, 2011.

[114] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78. IEEE Computer Society Press, 2012.

[115] P. F. Fischer, F. Hecht, and Y. Maday. A parareal in time semi-implicit approximation of the navier-stokes equations. In *Domain decomposition methods in science and engineering*, pages 433–440. Springer, 2005.

[116] H. Fu, C. He, B. Chen, Z. Yin, Z. Zhang, W. Zhang, T. Zhang, W. Xue, W. Liu, W. Yin, G. Yang, and X. Chen. 18.9pflopss nonlinear earthquake simulation on sunway taihulight: Enabling depiction of 18-hz and 8-meter scenarios. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 2:1–2:12, New York, NY, USA, 2017. ACM.

[117] H. Fu, J. Liao, N. Ding, X. Duan, L. Gan, Y. Liang, X. Wang, J. Yang, Y. Zheng, W. Liu, L. Wang, and G. Yang. Redesigning cam-se for peta-scale climate modeling performance and ultra-high resolution on sunway taihulight. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 1:1–1:12, New York, NY, USA, 2017. ACM.

[118] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, Jun 2016.

[119] J. D. Fulton. Generalized inverses of matrices over a finite field. *Discrete mathematics*, 21(1):23–29, 1978.

[120] G. Galli, T. Dunning, et al. Discovery in basic energy sciences: The role of computing at the extreme scale. In *ASCR Scientific Grand Challenges Workshop Series, Tech. Rep*, 2009.

[121] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 895–906, Nov 2014.

[122] M. Gamell, D. S. Katz, K. Teranishi, M. A. Heroux, R. F. Van der Wijngaart, T. G. Mattson, and M. Parashar. Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 346–355. IEEE, 2016.

[123] M. J. Gander. Analysis of the parareal algorithm applied to hyperbolic problems using characteristics. *Boletin de la Sociedad Espanola de Matemática Aplicada*, 42:21–35, 2008.

[124] M. J. Gander. 50 years of time parallel time integration. In *Multiple Shooting and Time Domain Decomposition Methods*, pages 69–113. Springer, 2015.

[125] M. J. Gander and E. Hairer. Nonlinear convergence analysis for the parareal algorithm. In *Domain decomposition methods in science and engineering XVII*, pages 45–56. Springer, 2008.

[126] M. J. Gander and E. Hairer. Analysis for parareal algorithms applied to hamiltonian differential equations. *Journal of Computational and Applied Mathematics*, 259:2–13, 2014.

[127] M. J. Gander and M. Neumüller. Analysis of a new space-time parallel multigrid algorithm for parabolic problems. *SIAM Journal on Scientific Computing*, 38(4):A2173–A2208, 2016.

[128] M. J. Gander and S. Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM Journal on Scientific Computing*, 29(2):556–578, 2007.

[129] M. J. Gander and S. Vandewalle. On the superlinear and linear convergence of the parareal algorithm. In *Domain decomposition methods in science and engineering XVI*, pages 291–298. Springer, 2007.

[130] J. Geiser and S. Güttel. Coupling methods for heat transfer and heat flow: Operator splitting and the parareal algorithm. *Journal of Mathematical Analysis and Applications*, 388(2):873 – 887, 2012.

[131] A. Geist. Supercomputing's monster in the closet. *IEEE Spectrum*, (3), 2016.

[132] V. Getov. Scientific grand challenges: Toward exascale supercomputing and beyond. *Computer*, 48(11):12–14, Nov 2015.

[133] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz. Extending stability beyond cpu millennium: A micron-scale atomistic simulation of kelvin-helmholtz instability. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 58:1–58:11, New York, NY, USA, 2007. ACM.

[134] A. Gluchoff. Artillerymen and mathematicians: Forest ray moulton and changes in american exterior ballistics, 1885–1934. *Historia Mathematica*, 38(4):506 – 547, 2011.

[135] D. Göddeke, M. Altenbernd, and D. Ribbrock. Fault-tolerant finite-element multigrid algorithms with hierarchically compressed asynchronous checkpointing. *Parallel Computing*, 49:117–135, 2015.

[136] H. H. Goldstine. *The Computer from Pascal to Von Neumann.* Princeton University Press, Princeton, NJ, USA, 1980.

[137] H. H. Goldstine and A. Goldstine. The electronic numerical integrator and computer (eniac). *IEEE Ann. Hist. Comput.*, 18(1):10–16, Mar. 1996.

[138] B. E. Golemba. *Human Computers: The Women in Aeronautical Research.* 1994.

[139] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed diskless checkpoint for large scale systems. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 63–72, Washington, DC, USA, 2010. IEEE Computer Society.

[140] L. B. Gomez, B. Nicolae, N. Maruyama, F. Cappello, and S. Matsuoka. Scalable reed-solomon-based reliable local storage for hpc applications on iaas clouds. In C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 313–324, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[141] A. Grama, V. Kumar, A. Gupta, and G. Karypis. *Introduction to parallel computing.* Pearson Education, 2003.

[142] D. A. Grier. *When computers were human.* Princeton University Press, 2013.

[143] R. Grout, H. Kolla, M. Minion, J. Bell, et al. Achieving algorithmic resilience for temporal integration through spectral deferred corrections. *Communications in Applied Mathematics and Computational Science*, 12(1):25–50, 2017.

[144] Q. Guan, N. DeBardeleben, S. Blanchard, and S. Fu. Addressing statistical significance of fault injection: Empirical studies of the soft error susceptibility.

International Journal of High Performance Computing and Networking, 10(4-5):436–452, 2017.

[145] P.-L. Guhur, E. Constantinescu, D. Ghosh, T. Peterka, and F. Cappello. Detection of silent data corruption in adaptive numerical integration solvers. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 592–602. IEEE, 2017.

[146] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. Failures in large scale systems: Long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 44:1–44:12, New York, NY, USA, 2017. ACM.

[147] G. Gurrala, A. Dimitrovski, S. Pannala, S. Simunovic, and M. Starke. Parareal in time for fast power system dynamic simulations. *IEEE Trans. Power Syst.*, 31(3):1820–1830, 2015.

[148] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[149] T. Haigh, P. M. Priestley, M. Priestley, and C. Rope. *ENIAC in action: Making and remaking the modern computer.* MIT press, 2016.

[150] D. Halacy. *Charles Babbage: father of the computer.* Crowell-Collier, 1970.

[151] E. Halley. 1705," astronomiae cometicae synopsis. *Philosoph. Transact. Royal Soc. London*, 24.

[152] E. Halley. Astronomical tables with precepts both in english and in latin for computing the places of the sun. *Moon, Planets, and Comets. London: W. Innys*, 1752.

[153] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[154] T. Hamada and K. Nitadori. 190 tflops astrophysical n-body simulation on a cluster of gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.

[155] Y. Hasegawa, J.-I. Iwata, M. Tsuji, D. Takahashi, A. Oshiyama, K. Minami, T. Boku, F. Shoji, A. Uno, M. Kurokawa, H. Inoue, I. Miyoshi, and M. Yokokawa. First-principles calculations of electron states of a silicon nanowire with 100,000 atoms on the k computer. In *Proceedings of 2011 International Conference for*

High Performance Computing, Networking, Storage and Analysis, SC '11, pages 1:1–1:11, New York, NY, USA, 2011. ACM.

[156] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan, M. Smelyanskiy, and P. Dubey. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 3–14, Piscataway, NJ, USA, 2014. IEEE Press.

[157] V. Heuveline, M. Schick, C. Webster, and P. Zaspel. Uncertainty quantification and high performance computing (dagstuhl seminar 16372). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[158] Hewlett Packard Enterprise. Exascale: A race to the future of HPC. Technical report, 2018. Technical white paper.

[159] J. Hines. ORNL Researcheres levarage GPU Tensor Cores to deliver unprecedented performance, 2018. link [Acccessed: 30.07.2018].

[160] A. P. Hinojosa, H.-J. Bungartz, and D. Pflüger. Scalable algorithmic detection of silent data corruption for high-dimensional pdes. In *Sparse Grids and Applications-Miami 2016*, pages 93–115. Springer, 2018.

[161] H. Hollerith. The electrical tabulating machine. *Journal of the Royal Statistical Society*, 57(4):678–689, 1894.

[162] R. Holmes, H. Strachan, C. Bellamy, and H. Bicheno. *The Oxford companion to military history*. Oxford University Press, USA, 2001.

[163] HPC4E Partners. About the use of exascale computers in Oil & Gas, Wind Energy and Biogas Combustion industries. Technical report, HPC4E, 2017.

[164] E. R. Hsu, J. D. Klemm, A. R. Kerlavage, D. Kusnezov, and W. A. Kibbe. Cancer moonshot data and technology team: Enabling a national learning healthcare system for cancer to unleash the power of data. *Clinical Pharmacology & Therapeutics*, 101(5):613–615, 2017.

[165] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage (TOS)*, 9(1):3, 2013.

[166] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. 2012.

[167] K.-H. Huang et al. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[168] F. D. Igual, M. Ali, A. Friedmann, E. Stotzer, T. Wentz, and R. A. van de Geijn. Unleashing the high-performance and low-power of multi-core dsps for general-purpose hpc. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2012.

[169] M. Iizuka and K. Ono. Influence of the phase accuracy of the coarse solver calculation on the convergence of the parareal method iteration for hyperbolic pdes. *Computing and Visualization in Science*, 2018.

[170] Y. Ishikawa. An Overview of Post-K Development. Slides used for invited talk the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, Tokyo, Japan, Jan. 28-31,2018.

[171] T. Ishiyama, K. Nitadori, and J. Makino. 4.45 pflops astrophysical n-body simulation on k computer: The gravitational trillion-body problem. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 5:1–5:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[172] K. Iskra, K. Yoshii, R. Gupta, and P. Beckman. Power management for exascale. *Mathematics and Computer Science Division Argonne National Laboratory*, 2012.

[173] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. De Supinski, and R. Eigenmann. Mcrengine: a scalable checkpointing system using data-aware aggregation and compression. *Scientific Programming*, 21(3-4):149–163, 2013.

[174] J. Jahns and S. H. Lee. *Optical Computing Hardware: Optical Computing*. Academic press, 2014.

[175] A. Jameson and J. Vassberg. Computational fluid dynamics for aerodynamic design-its current and future impact. In *39th Aerospace Sciences Meeting and Exhibit*, page 538.

[176] N. Jasika, N. Alispahic, A. Elma, K. Ilvana, L. Elma, and N. Nosovic. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 1811–1815, May 2012.

[177] S. Jin and Z. Xin. The relaxation schemes for systems of conservation laws in arbitrary space dimensions. *Communications on pure and applied mathematics*, 48(3):235–276, 1995.

[178] W. M. Jones, J. T. Daly, and N. DeBardeleben. Application monitoring and checkpointing in hpc: Looking towards exascale systems. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, pages 262–267, New York, NY, USA, 2012. ACM.

[179] J. Jordan, T. Ippen, M. Helias, I. Kitayama, M. Sato, J. Igarashi, M. Diesmann, and S. Kunkel. Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers. *Frontiers in neuroinformatics*, 12:2, 2018.

[180] E. C. Joseph, R. Sorensen, S. Conway, and K. Monroe. Analysis of the Characteristics and Development Trends of the Next-Generation of Supercomputers in Foreign Countries. Technical report, IDC on behalf of RIKEN Advanced Institute for Computational Science, june 2016.

[181] G. R. Joubert, H. Leather, and M. Parsons. *Parallel Computing: On the Road to Exascale*, volume 27. IOS Press, 2016.

[182] D. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, and B. Wohlmuth. Multiphysics simulations: Challenges and opportunities. 27, 02 2013.

[183] M. A. Khaleel, G. M. Johnson, and W. M. Washington. Scientific grand challenges: Challenges in climate change science and the role of computing at the extreme scale. Technical report, Pacific Northwest National Laboratory (PNNL), Richland, WA (US), 2009.

[184] D. B. Kothe. Science prospects and benefits with exascale computing. Technical report, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2007.

[185] P. Kovatch, M. Ezell, and R. Braby. The malthusian catastrophe is upon us! are the largest hpc machines ever up? In *European Conference on Parallel Processing*, pages 211–220. Springer, 2011.

[186] A. Kozhevnikov, A. G. Eguiluz, and T. C. Schulthess. Toward first principles electronic structure simulations of excited states and strong correlations in nano- and materials science. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, Nov 2010.

[187] A. Kreienbuehl, A. Naegel, D. Ruprecht, R. Speck, G. Wittum, and R. Krause. Numerical simulation of skin transport using parareal. *Computing and visualization in science*, 17(2):99–108, 2015.

[188] K. Kumahata, K. Minami, and N. Maruyama. High-performance conjugate gradient performance improvement on the k computer. *The International Journal of High Performance Computing Applications*, 30(1):55–70, 2016.

[189] J. Langou, Z. Chen, J. J. Dongarra, and G. Bosilca. Disaster survival guide in petascale computing. In *Petascale Computing: Algorithms and Applications*, pages 263–288. Chapman and Hall/CRC, 2007.

[190] R. J. LeVeque. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[191] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. Modeling soft-error propagation in programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE*, 2018.

[192] J.-L. Lions, Y. Maday, and G. Turinici. Résolution d'edp par un schéma en temps pararéel. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 332(7):661–668, 2001.

[193] J.-L. Lions, Y. Maday, and G. Turinici. Résolution d'edp par un schéma en temps pararéel. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 332(7):661–668, 2001.

[194] Q. Liu, C. Jung, D. Lee, and D. Tiwari. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 20. IEEE Press, 2016.

[195] Q. Liu and W. Luk. Heterogeneous systems for energy efficient scientific computing. In O. C. S. Choy, R. C. C. Cheung, P. Athanas, and K. Sano, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, pages 64–75, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[196] T. Loderer, V. Heuveline, and R. Lohner. The parareal algorithm as a new approach for numerical integration of odes in real-time simulations in automotive industry. *PAMM*, 14(1):1027–1030, 2014.

[197] B. Lojek. *History of semiconductor engineering*. Springer, 2007.

[198] M. Luisier, T. B. Boykin, G. Klimeck, and W. Fichtner. Atomistic nanoelectronic device engineering with sustained performances up to 1.44 pflop/s. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 2:1–2:11, New York, NY, USA, 2011. ACM.

[199] P. Lynch. *The emergence of numerical weather prediction: Richardson's dream*. Cambridge University Press, 2006.

[200] Y. Maday and G. Turinici. A parareal in time procedure for the control of partial differential equations. *Comptes Rendus Mathematique*, 335(4):387–392, 2002.

[201] D. Maidment and L. Mays. *Applied Hydrology*. McGraw-Hill series in water resources and environmental engineering. Tata McGraw-Hill Education, 1988.

[202] M. Manasse, C. Thekkath, and A. Silverberg. A reed-solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage.

[203] V. Marjanović, J. Gracia, and C. W. Glass. Performance modeling of the hpcg benchmark. In S. A. Jarvis, S. A. Wright, and S. D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 172–192, Cham, 2015. Springer International Publishing.

[204] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter. Nvidia tensor core programmability, performance & precision. *arXiv preprint arXiv:1803.04014*, 2018.

[205] H. Markram, K. Meier, T. Lippert, S. Grillner, R. Frackowiak, S. Dehaene, A. Knoll, H. Sompolinsky, K. Verstreken, J. DeFelipe, S. Grant, J.-P. Changeux, and A. Saria. Introducing the human brain project. *Procedia Computer Science*, 7:39 – 42, 2011. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

[206] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 610–621, June 2014.

[207] T. P. Mathew, M. Sarkis, and C. E. Schaerer. Analysis of block parareal preconditioners for parabolic optimal control problems. *SIAM Journal on Scientific Computing*, 32(3):1180–1200, 2010.

[208] H. W. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. *The TOP500: History, Trends, and Future Directions in High Performance Computing.* Chapman & Hall/CRC, 1st edition, 2014.

[209] H. W. Meuer, E. Strohmaier, J. Dongarra, H. D. Simon, and M. Meuer. TOP500, 2017. link [Acccessed: 04.06.2018].

[210] S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer. *IEEE Transactions on Device and Materials Reliability*, 12(2):445–454, 2012.

[211] M. Minion. A hybrid parareal spectral deferred corrections method. *Communications in Applied Mathematics and Computational Science*, 5(2):265–301, 2011.

[212] M. L. Minion et al. A hybrid parareal spectral deferred corrections method. *Communications in Applied Mathematics and Computational Science*, 5(2):265–301, 2010.

[213] W. L. Miranker and W. Liniger. Parallel methods for the numerical integration of ordinary differential equations. *Mathematics of Computation*, 21(99):303–320, 1967.

[214] S. Mittal and J. S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[215] S. Mittal and J. S. Vetter. A survey of techniques for modeling and improving reliability of computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1226–1238, 2016.

[216] A. Mohebbi and M. Dehghan. High-order compact solution of the one-dimensional heat and advection–diffusion equations. *Applied Mathematical Modelling*, 34(10):3071–3084, 2010.

[217] K. Mohror, A. Moody, G. Bronevetsky, and B. R. de Supinski. Detailed modeling and evaluation of a scalable multilevel checkpointing system. *IEEE Transactions on Parallel and Distributed Systems*, 25(9):2255–2263, 2014.

[218] D. Monroe. Neuromorphic computing gets ready for the (really) big time. *Communications of the ACM*, 57(6):13–15, 2014.

[219] A. Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.

[220] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[221] T. Muranushi, H. Hotta, J. Makino, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, N. Hosono, Y. Maruyama, H. Inoue, H. Yashiro, and Y. Nakamura. Simulations of below-ground dynamics of fungi: 1.184 pflops attained by automated generation and autotuning of temporal blocking codes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’16, pages 3:1–3:11, Piscataway, NJ, USA, 2016. IEEE Press.

[222] P. Mycek, A. Contreras, O. Le Maître, K. Sargsyan, F. Rizzi, K. Morris, C. Safta, B. Debusschere, and O. Knio. A resilient domain decomposition polynomial chaos solver for uncertain elliptic pdes. *Computer Physics Communications*, 216:18–34, 2017.

[223] N. Katsuya, J. Russell, HPCwire. France’s CEA and Japan’s RIKEN to Partner on ARM and Exascale, 2017. link [Acccessed: 07.06.2018].

[224] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, S. L. Scott, et al. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *2008 8th International Symposium on Cluster Computing and the Grid (CCGRID*, pages 783–788. IEEE, 2008.

[225] R. B. Neale, C.-C. Chen, A. Gettelman, P. H. Lauritzen, S. Park, D. L. Williamson, A. J. Conley, R. Garcia, D. Kinnison, J.-F. Lamarque, et al. Description of the ncar community atmosphere model (cam 5.0). *NCAR Tech. Note NCAR/TN-486+ STR*, 1(1):1–12, 2010.

[226] I. Newton. 1687. *Philosophiae naturalis principia mathematica*, 3, 1995.

[227] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[228] A. S. Nielsen. Feasibility study of the parareal algorithm. Diss. msc thesis, Technical University of Denmark, 2012. IMM-134.

[229] A. S. Nielsen, G. Brunner, and J. S. Hesthaven. Communication-aware adaptive parareal with application to a nonlinear hyperbolic system of partial differential equations. *Journal of Computational Physics*, 371:483 – 505, 2018.

[230] A. S. Nielsen and C. Cantwell. Llama Repository, 2018. link [Acccessed: 28.07.2018].

[231] A. S. Nielsen and J. S. Hesthaven. Fault tolerance in the parareal method. In *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, pages 1–8. ACM, 2016.

[232] J. Nievergelt. Parallel methods for integrating ordinary differential equations. *Communications of the ACM*, 7(12):731–733, 1964.

[233] Office of Advanced Scientific Computing Research, Department of Energy, Office of Science. The Exascale Effect: the Benefits of Supercomputing Investment for U.S. Industry. Technical report, Council on Competitiveness, 2014.

[234] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech. Experimental and analytical study of xeon phi reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 28. ACM, 2017.

[235] C. Pachajoa and W. N. Gansterer. On the resilience of conjugate gradient and multigrid methods to node failures. In D. B. Heras, L. Bougee, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer, editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 569–580, Cham, 2018. Springer International Publishing.

[236] G. Pages, O. Pironneau, and G. Sall. The parareal algorithm for american options. *Comptes Rendus Mathematique*, 354(11):1132–1138, 2016.

[237] T. Palmer. Climate forecasting: Build high-resolution global climate models. *Nature News*, 515(7527):338, 2014.

[238] D. S. Papailiopoulos and A. G. Dimakis. Locally repairable codes. *IEEE Transactions on Information Theory*, 60(10):5843–5855, 2014.

[239] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl-a portable implementation of the high-performance linpack benchmark for distributed-memory computers, version 2.0. *Innovative Computing Laboratory, University of Tennessee*, 10, 2008.

[240] E. T. Phipps, J. J. Hu, H. C. Edwards, and C. G. Webster. Realizing exascale performance for uncertainty quantification. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States); Sandia National Laboratories, Livermore, CA, 2013.

[241] J. S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using raid techniques. In *srds*, page 76. IEEE, 1996.

[242] J. S. Plank, K. Greenan, E. Miller, and W. Houston. Gf-complete: A comprehensive open source library for galois field arithmetic. *University of Tennessee, Tech. Rep. UT-CS-13-703*, 2013.

[243] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[244] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[245] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *SIGARCH Comput. Archit. News*, 42(3):13–24, June 2014.

[246] F. Qiao, W. Zhao, X. Yin, X. Huang, X. Liu, Q. Shu, G. Wang, Z. Song, X. Li, H. Liu, G. Yang, and Y. Yuan. A highly effective global surface wave numerical simulation with ultra-high resolution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 5:1–5:11, Piscataway, NJ, USA, 2016. IEEE Press.

[247] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[248] A. Randles and E. Kaxiras. Parallel in time approximation of the lattice boltzmann method for laminar flows. *Journal of Computational Physics*, 270:577–586, 2014.

[249] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[250] J. M. Reynolds-Barredo, D. E. Newman, R. Sanchez, D. Samaddar, L. A. Berry, and W. R. Elwasif. Mechanisms for the convergence of time-parallelized, parareal turbulent plasma simulations. *Journal of Computational Physics*, 231(23):7851–7867, 2012.

[251] Rick Stevens, Argonne National Laboratory. Lighting the way to Exascale Precision Medicine, 2018. link [Acccessed: 11.06.2018].

[252] S. Ristov, R. Prodan, M. Gusev, and K. Skala. Superlinear speedup in hpc systems: Why and when? In *Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on*, pages 889–898. IEEE, 2016.

[253] P. L. Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[254] R. Rojas and U. Hashagen. *The first computers: History and architectures*. MIT press, 2002.

[255] D. Rossinelli, B. Hejazialhosseini, P. Hadjidoukas, C. Bekas, A. Curioni, A. Bertsch, S. Futral, S. J. Schmidt, N. A. Adams, and P. Koumoutsakos. 11 pflop/s simulations of cloud cavitation collapse. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 3:1–3:13, New York, NY, USA, 2013. ACM.

[256] D. Rossinelli, Y.-H. Tang, K. Lykov, D. Alexeev, M. Bernaschi, P. Hadjidoukas, M. Bisson, W. Joubert, C. Conti, G. Karniadakis, M. Fatica, I. Pivkin, and P. Koumoutsakos. The in-silico lab-on-a-chip: Petascale and high-throughput simulations of microfluidics at cell resolution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 2:1–2:12, New York, NY, USA, 2015. ACM.

[257] J. Rudi, A. C. I. Malossi, T. Isaac, G. Stadler, M. Gurnis, P. W. J. Staar, Y. Ineichen, C. Bekas, A. Curioni, and O. Ghattas. An extreme-scale implicit solver for complex pdes: Highly heterogeneous flow in earth's mantle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 5:1–5:12, New York, NY, USA, 2015. ACM.

[258] D. Ruprecht. Convergence of parareal with spatial coarsening. *PAMM*, 14(1):1031–1034, 2014.

[259] D. Ruprecht. Wave propagation characteristics of parareal. *Computing and Visualization in Science*, 19(1):1–17, 2018.

[260] D. Ruprecht and R. Krause. Explicit parallel-in-time integration of a linear acoustic-advection system. *Computers & Fluids*, 59:72–83, 2012.

[261] D. Ruprecht, R. Speck, M. Emmett, M. Bolten, and R. Krause. Extreme-scale space-time parallelism.

[262] D. Samaddar, D. E. Newman, and R. Sánchez. Parallelization in time of numerical simulations of fully-developed plasma turbulence using the parareal algorithm. *Journal of Computational Physics*, 229(18):6558–6573, 2010.

[263] J. C. Sancho, F. Petrini, G. Johnson, and E. Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 58. IEEE, 2004.

[264] M. R. Sancho, N. Alexandrova, and M. Gonzalez. Addressing hpc skills shortages with parallel computing mooc. In *2015 International Conference on Interactive Collaborative and Blended Learning (ICBL)*, pages 86–93, Dec 2015.

[265] P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, page 4. ACM, 2013.

[266] M. Sarkis, C. E. Schaerer, and T. Mathew. Block diagonal parareal preconditioner for parabolic optimal control problems. In *Domain decomposition methods in science and engineering XVII*, pages 409–416. Springer, 2008.

[267] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

[268] D. Schneider. Could supercomputing turn to signal processors (again)? *IEEE Spectrum*, 49(10):13–14, October 2012.

[269] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, Oct 2010.

[270] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *FAST*, volume 7, pages 1–16, 2007.

[271] B. Schroeder, R. Lagisetty, and A. Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, 2016.

[272] R. F. Service. China's planned exascale computer threatens summit's position at the top. *Science*, 359(6376):618–618, 2018.

[273] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein. CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. *CoRR*, abs/1708.02030, 2017.

[274] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the international conference on Supercomputing*, pages 152–161. ACM, 2011.

[275] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 69–78. ACM, 2012.

[276] A. Shehabi, S. J. Smith, D. A. Sartor, R. E. Brown, M. Herrlin, J. G. Koomey, E. R. Masanet, N. Horner, I. L. Azevedo, and W. Lintner. United states data center energy usage report. Technical report, 06/2016 2016.

[277] E. Shein. Combating cancer with data. *Commun. ACM*, 60(5):10–12, Apr. 2017.

[278] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 3:1–3:11, New York, NY, USA, 2011. ACM.

[279] C.-W. Shu. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. In *Advanced numerical approximation of nonlinear hyperbolic equations*, pages 325–432. Springer, 1998.

[280] L. M. Silva and J. G. Silva. An experimental study about diskless checkpointing. In *Euromicro Conference, 1998. Proceedings. 24th*, volume 1, pages 395–402. IEEE, 1998.

[281] J. Smagorjnsky. The beginnings of numerical weather prediction and general circulation modeling: early recollections. In *Advances in Geophysics*, volume 25, pages 3–37. Elsevier, 1983.

[282] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[283] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. Minion, M. Winkel, and P. Gibbon. A massively space-time parallel n-body solver. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 92. IEEE Computer Society Press, 2012.

[284] M. A. Sprague, S. Boldyrev, P. Fischer, R. Grout, W. I. Gustafson Jr, and R. Moser. Turbulent flow simulation at the exascale: Opportunities and challenges workshop: August 4-5, 2015, washington, dc. Technical report, NREL (National Renewable Energy Laboratory (NREL), Golden, CO (United States)), 2017.

[285] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *ACM SIGPLAN Notices*, volume 50, pages 297–310. ACM, 2015.

[286] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi. Feng shui of supercomputer memory: positional effects in dram and sram faults. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 22. ACM, 2013.

[287] A. Srinivasan and N. Chandra. Latency tolerance through parallelization of time in scientific applications. *Parallel Computing*, 31(7):777–796, 2005.

[288] G. A. Staff and E. M. Rønquist. Stability of the parareal algorithm. In *Domain decomposition methods in science and engineering*, pages 449–456. Springer, 2005.

[289] J. Steiner, D. Ruprecht, R. Speck, and R. Krause. Convergence of parareal for the navier-stokes equations depending on the reynolds number. In *Numerical Mathematics and Advanced Applications-ENUMATH 2013*, pages 195–202. Springer, 2015.

[290] M. Stoyanov and C. Webster. Numerical analysis of fixed point algorithms in the presence of hardware faults. *SIAM Journal on Scientific Computing*, 37(5):C532–C553, 2015.

[291] E. Strohmaier. 20 years supercomputer market analysis. 2005.

[292] E. Strohmaier, H. W. Meuer, J. Dongarra, and H. D. Simon. The top500 list and progress in high-performance computing. *Computer*, 48(11):42–49, 2015.

[293] O. Subasi, G. Kestor, and S. Krishnamoorthy. Toward a general theory of optimal checkpoint placement. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 464–474, Sept 2017.

[294] O. Subasi, G. Yalcin, F. Zyulkyarov, O. Unsal, and J. Labarta. Designing and modelling selective replication for fault-tolerant hpc applications. In *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*, pages 452–457. IEEE, 2017.

[295] X. Tang, J. Zhai, B. Yu, W. Chen, and W. Zheng. Self-checkpoint: An in-memory checkpoint method using less space and its practice on fault-tolerant hpl. In *ACM SIGPLAN Notices*, volume 52, pages 401–413. ACM, 2017.

[296] K. Teranishi and M. Gamell. Fenix Repository, 2018. link [Acccessed: 28.07.2018].

[297] W. C. Thacker. Some exact solutions to the nonlinear shallow-water wave equations. *Journal of Fluid Mechanics*, 107:499–508, 1981.

[298] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 331–342. IEEE, 2015.

[299] K. H. Tsoi and W. Luk. Axel: A heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 115–124, New York, NY, USA, 2010. ACM.

[300] U.S. Department of Energy. Secretary of Energy Rick Perry Announces $1.8 Billion Initiative for New Supercomputers, 2018. link [Acccessed: 06.06.2018].

[301] N. H. Vaidya. A case for two-level distributed recovery schemes. In *ACM SIGMETRICS Performance Evaluation Review*, volume 23, pages 64–73. ACM, 1995.

[302] S. Vandewalle and E. Van de Velde. Space-time concurrent multigrid waveform relaxation. *Annals of Numerical Mathematics*, 1(1-4):335–346, 1994.

[303] A. Weber. The USC-SIPI Image Database, 2018. link [Acccessed: 02.08.2018].

[304] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 375–382. IEEE, 2014.

[305] S. B. Wicker and V. K. Bhargava. An introduction to reed-solomon codes.

[306] M. R. Williams. *A History of Computing Technology, 2Nd Edition.* IEEE Computer Society Press, Los Alamitos, CA, USA, 2nd edition, 1997.

[307] C. Wilson. Clairaut's calculation of the eighteenth-century return of halley's comet. *Journal for the History of Astronomy*, 24(1-2):1–15, 1993.

[308] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon. A massively parallel, multi-disciplinary barnes–hut tree code for extreme-scale n-body simulations. *Computer physics communications*, 183(4):880–889, 2012.

[309] R. L. Winslow, N. Trayanova, D. Geman, and M. I. Miller. Computational medicine: translating models to clinical care. *Science translational medicine*, 4(158):158rv11–158rv11, 2012.

[310] C.-K. Wu and E. Dawson. Existence of generalized inverse of linear transformations over finite fields. *Finite Fields and Their Applications*, 4(4):307–315, 1998.

[311] P. Wu, N. DeBardeleben, Q. Guan, S. Blanchard, J. Chen, D. Tao, X. Liang, K. Ouyang, and Z. Chen. Silent data corruption resilient two-sided matrix factorizations. In *ACM SIGPLAN Notices*, volume 52, pages 415–427. ACM, 2017.

[312] S.-L. Wu. Convergence analysis of some second-order parareal algorithms. *IMA Journal of Numerical Analysis*, 35(3):1315–1341, 2015.

[313] S.-L. Wu and T. Zhou. Parareal algorithms with local time-integrators for time fractional differential equations. *Journal of Computational Physics*, pages –, 2018.

[314] H. Xiao and E. Aubanel. Scheduling of tasks in the parareal algorithm for heterogeneous cloud platforms. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1440–1448. IEEE, 2012.

[315] Y. Xing and C.-W. Shu. High-order finite volume weno schemes for the shallow water equations with dry states. *Advances in Water Resources*, 34(8):1026–1038, 2011.

[316] Xinhua News Agency. Sample machine for China's exascale supercomputer operational this year , 2018. link [Acccessed: 06.06.2018].

[317] C. Yang, W. Xue, H. Fu, H. You, X. Wang, Y. Ao, F. Liu, L. Gan, P. Xu, L. Wang, G. Yang, and W. Zheng. 10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 6:1–6:12, Piscataway, NJ, USA, 2016. IEEE Press.

[318] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[319] Y. Yuan, Y. Wu, Q. Wang, G. Yang, and W. Zheng. Job failures in high performance computing systems: A large-scale empirical study. *Computers & Mathematics with Applications*, 63(2):365–377, 2012.

[320] G. Zheng, X. Ni, and L. V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[321] G. Zheng, L. Shi, and L. V. Kalé. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *Cluster Computing, 2004 IEEE International Conference on*, pages 93–103. IEEE, 2004.

Allan Svejstrup Nielsen

Contact

Chemin de la Venoge 2
1028 Preverenges, Switzerland
Mobile: +41 78 902 52 48
Email: svejstrup@ieee.org

Languages

Danish C2, English C2, Mandarin Chinese C1, French B1.

Programming

Matlab, C, C++, MPI, Pthreads, CUDA, OpenMP, Python, OpenCL.

Education

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Doctor of Philosophy in Computational Mathematics **2013 – 2018**

- Student of Prof. Jan Hesthaven at Chair of Computational Mathematics and Simulation Science
- Developed and published novel contributions in the field of exascale computational science.
- GPA 5.5/6.0. Coursework on Deep Learning, Big Data, Neural Networks and Parallel HPC.

Fudan University, Shanghai, China

Laurits Andersen Scholar **2012 – 2013**

- Studying Mandarin Chinese at the Cultural Exchange School. Reached level HSK4.

Technical University of Denmark, Copenhagen, Denmark

Master of Science in Engineering, Applied Mathematics **2010 – 2012**

- GPA 11.8/12.0. Graduated with Honors. Coursework on Entrepreneurship, Project Management, High-Performance Computing, Multivariate Statistics, Scientific Computing, Numerical Analysis, Stochastic Calculus. Exchange student for 5 months at National University of Singapore, 2010.

Bachelor of Science in Engineering, Physics and Nanotechnology **2007 – 2010**

- GPA 10.3/12.0. Coursework on Quantum Mechanics, Integrated Circuits and Microelectromechanical Systems. Exchange student for 5 months at U. of Illinois - Urbana Champaign, 2009.

Community Services

President, SIAM @ EPFL **2015 – 2017**

Founding president of the EPFL Chapter of the Society for Industrial and Applied Mathematics. Created an annual workshop on entrepreneurship, and a pitch-your-research competition for students.

Volunteer, Chinese Students & Scholars Association Lausanne **2014 – 2015**

Gave presentations in mandarin chinese on life in Lausanne and activities at the sports-center to newly arrived chinese EPFL students. Assisted with organizing community activities and events.

Board Chairman, Viggo Jarls Kollegium **2011 – 2012**

Served as Board Chairman of the Resident Association at Viggo Jarls Kollegium. Organized social activities and raised funds among residents to purchase new sports equipment for the Kollegium.

Guide and Mentor, DTU International Affairs **2011 – 2012**

Mentor in the mentor-mentee program for international students commencing their studies at DTU. Received the highest rating among all 40 guides as evaluated by the students during both semesters.

Team Leader, ESSIM 2011 **Summer 2011**

Selected to represent DTU by the European Consortium for Mathematics in Industry. Led research project and presented results to 120 fellow students, faculty and industry participants.

Honors and Awards

SIAM Student Chapter Certificate of Recognition **Spring 2018**

Awarded for outstanding service and contributions to the EPFL Chapter of SIAM.

Award for Excellence in Teaching **Spring 2015**

Prize presented annually by the Doctoral School of Mathematics. Awarded for excellence in supervising student projects, and for contributions in the teaching of Analysis for Physicists.

The Gauß-Allianz Prize **Fall 2012**

Received for best conference talk at the 2nd Multicore-Challenge Conference, HLRS, Germany.

Laurits Andersen Scholar **Fall 2011**

Selected as one of only five young Danish graduates to receive the DKK 120.000 stipend presented annually by the Laurits Andersen foundation to promote and develop Sino-Danish relations.

Various Merit-Based Scholarships **2009 – 2010**

Raised DKK 75.000 in scholarships to fund studies at UIUC in the USA and at NUS in Singapore.

Teaching Activities

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland **2013 – 2018**

Teaching assistant the the EPFL Department of Mathematics. Responsibilities included preparing exercises, helping students at exercise sessions, office hours, preparing exam questions and projects, correcting exams, grading exams and giving lectures in the absence of the professor.

- MATH-458 Programming Concepts in Scientific Computing, Fall ’15, ’16 and ’17.
- MATH-454 Parallel and High-Performance Computing, Spring ’14, ’15, ’16 and ’17.
- MATH-100 Analysis I for Physicists, Fall ’14.
- MATH-459 Numerical Methods for Conservation Laws, Fall ’13.

Technical University of Denmark, Copenhagen, Denmark **2011 – 2012**

Teaching assistant at DTU Compute. Tutor at problem solving sessions.

- Course 01005 Advanced Engineering Mathematics 1, Fall ’11 and Spring ’12

List of Conference Talks

CAAP with Application to a Nonlinear Hyperbolic System of PDEs at *SIAM Parallel Processing 18*, Waseda University, Tokyo Japan, March 2018.

CAAP with Application to a Nonlinear Hyperbolic System of PDEs at *6th Workshop on Parallel-in-Time Integration Methods*, Congressi Stefano Franscini, Monte Verità, Switzerland, Oct. 2017.

Multilevel Diskless Checksum Checkpoints for Automated Application Recovery at *Platform for Advanced Scientific Computing 17*, Palazzo dei Congressi, Lugano, Switzerland, June 2017.

Fault Tolerance in the Parareal Method at *SIAM Computational Science and Engineering 17*, Atlanta Hilton Hotel, Georgia, USA, March 2017.

Space-Time Parallelism for Hyperbolic PDEs at *Platform for Advanced Scientific Computing 16*, SwissTech Convention Center, Lausanne, Switzerland, June 2016.

Fault Tolerance in the Parareal Method at *ACM Workshop on Fault-Tolerance for HPC at Extreme Scale, 25th ACM Symposium on High-Performance Parallel and Distributed Computing*, Kyoto International Community House, Kyoto, Japan, May 2016.

Parareal for Multi-Layered Parallelism at *Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing*, HFT Stuttgart, Baden-Württemberg, Germany, Sept. 2012.

List of Publications

C.D. Cantwell and A.S. Nielsen “A minimally intrusive Low-memory approach to Resilience for existing Transient Solvers” in *Journal of Scientific Computing*, Jul. 2018.

A.S. Nielsen, G. Brunner and J.S. Hesthaven “Communication-aware Adaptive Parareal with application to a Nonlinear Hyperbolic System of Partial Differential Equations” in *Journal of Computational Physics*, 371:483 505, 2018.

A.S. Nielsen and J.S. Hesthaven “Fault Tolerance in the Parareal Method” in *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale* at the 25th ACM Symposium on High-Performance Parallel and Distributed Computing, 2016.

F. Chen, J.S. Hesthaven, Y. Maday, and A.S. Nielsen “An Adjoint Approach for Stabilizing the Parareal Method”, *Technical Report*, 2015.

S.L. Glimberg, A.P. Engsig-Karup, A.S. Nielsen and B. Dammann “Development of software components for heterogeneous many-core architectures” in *Designing Scientific Applications on GPUs*, Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, 2013

A.P. Engsig-Karup, S.L. Glimberg, A.S. Nielsen and O. Lindberg “Fast hydrodynamics on heterogeneous many-core hardware” in *Designing Scientific Applications on GPUs*, Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, 2013

A.S. Nielsen “Feasibility study of the parareal algorithm”. *Diss. MSc thesis*, Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2012.

A.S. Nielsen, A.P. Engsig-Karup and B. Dammann, “Parallel Programming using OpenCL on Modern Architectures” *DTU IMM Technical Report No. 2012-05*, Technical University of Denmark, 2012.

L.M. Fischer, A.S. Nielsen, S. Dohn, M. Tenje, A. Boisen “An Electrochemical-Cantilever Hybrid Sensor for Metal Ions” *IEEE Sensors 2010 proceedings*, pp. 913-917, 2010.