Performance Prediction and Race Detection in Message-Passing Parallel Applications

THÈSE N^{O} 4274 (2009)

PRÉSENTÉE LE 6 FÉVRIER 2009

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES PÉRIPHÉRIQUES
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Basile SCHÄLI

acceptée sur proposition du jury:

Prof. R. Guerraoui, président du jury
Prof. R. Hersch, directeur de thèse
Dr R. Gruber, rapporteur
Prof. P. Kropf, rapporteur
Dr S. Ur, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2009

Abstract

The combination of low cost clusters and multicore processors lowers the barrier for accessing massive amounts of computing power. As computational sciences advance, the use of *in silico* simulations to complement *in vivo* experiments promises parallel programming a bright future in multiple scientific fields. It is therefore increasingly important to develop tools helping developers to write efficient and bug-free parallel applications.

This thesis focuses on performance prediction and advanced testing tools for distributed memory message-passing parallel applications. The tools have been implemented within the Dynamic Parallel Schedules (DPS) parallelization framework. They have also been partly adapted to applications written using the Message Passing Interface (MPI) standard.

The first part presents a parallel application simulator which has been integrated into the DPS framework. We identified a small set of processing and networking parameters that characterize the hardware platform on which the application is running. After parameterizing the hardware platform, the running time of parallel applications can be predicted using direct execution without requiring any change to the application source code. We propose a partial direct execution technique that reduces the execution time and memory consumption of the simulation. Using partial direct execution, the simulation is no longer tied to the platform to be simulated. Simulations may thus run on a desktop computer rather than on the target parallel machine. The proposed parameterization of the application and of the hardware properties enable using the simulator to study the sensitivity of a parallel application to various operating conditions such as the data subdivision granularity, the adopted parallelization strategy and the underlying hardware platform properties. The proposed simulator helps developers identifying the factors having the largest impact on their application's performance, and determining the most suitable cluster hardware configuration.

Speed should not come at the expense of correctness. Since improving parallelization efficiency often requires loosening synchronizations or implementing more complex communication patterns, developers need to ensure that their changes do not introduce potential message races or deadlocks. Deadlocks and message races are common sources of problems in parallel applications and stem from the fact that the delivery of messages from different sources is not deterministically ordered. This non-determinism makes such synchronization errors hard to reproduce and debug.

The second part of the thesis presents methods for uncovering potential deadlocks and message races by taking advantage of the flow graph structures and checkpointing capabilities of DPS. We developed a debugger for DPS applications that displays an instantaneous graphical view of the global computation state and is able to control the ordering of message delivery in order to explicitly test specific orderings. The number of possible orderings explodes when the number of messages sent by the application increases. Manual testing can only cover a tiny fraction of possible executions. Therefore, we use the simulator's ability to control the execution of a parallel application, in order to automatically detect deadlocks and message races. A first method for reducing the number of orderings to be tested relies on a partial-order reduction of the search space and on the decomposition of the application execution into independently testable subparts. This method relies on a static analysis of an execution trace of the application, and can therefore only be applied to parallel applications that produce a fixed set of messages, i.e. applications producing the same messages for all delivery orderings. In order to overcome this limitation, we propose an approach relying on the dynamic construction of a state graph expressing possible executions. Both methods reduce the testing costs by several orders of magnitude, and can be combined to further improve the results. Nevertheless, testing durations may remain prohibitive for longer running applications. We therefore also define algorithms generating subsets of possible orderings that are likely to reveal erroneous executions.

In the recent years, the MPI standard has emerged as the *de facto* standard for writing message-passing parallel applications. The final part of this thesis therefore focuses on adapting the aforementioned parallel application testing concepts to MPI applications. We first describe the extension of our work on visualizing the execution of parallel applications. We then discuss the limits and the benefits of using partial-order execution graphs to describe MPI application executions, and show that our dynamic message-passing state graph construction approach can be successfully applied.

Keywords : Message-passing parallel applications, performance prediction, message race, testing, partial-order reduction, debugging, Dynamic Parallel Schedules, Message Passing Interface

Résumé

La combinaison de clusters à bas coûts et de processeurs multicoeurs expose un public de plus en plus large à une puissance de calcul considérable. Avec l'avance des sciences computationnelles, l'utilisation de simulations *in silico* pour compléter les expériences *in vivo* promettent à la programmation parallèle un futur radieux dans un nombre croissants de domaines scientifiques. Il est donc de plus en plus important de développer des outils pour aider les développeurs à écrire des applications parallèles efficaces et correctes.

Cette thèse se concentre sur la prédiction de performance et sur des outils de tests avancés pour les applications parallèles à mémoire distribuée communiquant par envoi de messages. Ces outils ont été implémentés dans la framework de programmation parallèle Dynamic Parallel Schedules (DPS). Ils ont aussi été partiellement adaptés aux applications utilisant le standard Message Passing Interface (MPI).

La première partie présente un simulateur d'applications parallèles intégré dans DPS. Nous avons identifié un petit ensemble de paramètres caractérisant les performances de calcul et de communication de la plateforme matérielle sur laquelle tourne l'application. Après avoir paramétrisé le matériel, le temps d'exécution des applications parallèles peut être prédit sans modifier le code de l'application en utilisant de l'exécution directe. Nous proposons une technique d'exécution directe partielle réduisant la durée de la simulation et sa consommation mémoire. En utilisant l'exécution directe partielle, la simulation n'est plus liée à la plateforme devant être simulée. Les simulations peuvent donc être effectuées sur un ordinateur de bureau plutôt que sur la machine parallèle cible. La paramétrisation de l'application et des caractéristiques matérielles permettent d'utiliser le simulateur pour étudier la sensibilité d'une application parallèle aux différentes conditions d'opérations, telles que la granularité de subdivision, la stratégie de parallélisation et la vitesse des calculs ou des communications. Le simulateur proposé peut donc aider les développeurs à identifier les facteurs ayant le plus d'impact sur les performances de leur application, et à déterminer quelle machine parallèle est la plus appropriée.

L'amélioration de la vitesse d'exécution ne doit pas induire des erreurs dans le programme.

L'efficacité de la parallélisation est souvent améliorée en relâchant certaines synchronizations ou en implémentant des patterns de communications plus complexes. Les développeurs doivent donc s'assurer que leurs changements n'introduisent pas des interblockages (deadlocks) ou des situations de compétitions (message races). Ces deux problèmes sont fréquents dans les programmes parallèles et sont dûs au fait que des messages provenant de différentes sources n'arrivent pas dans un ordre déterministe. Ce non-déterminisme rend ces erreurs de synchronizations difficiles à reproduire et à corriger.

La deuxième partie de cette thèse présente des méthodes provoquant et détectant les deadlocks et message races potentiels en tirant parti du graphe de flux et des capacités de checkpointing des applications DPS. Nous présentons un débuggeur capable de représenter graphiquement l'état d'avancement de l'exécution d'une application DPS. Un contrôle sur l'ordonnancement des messages permet de tester spécifiquement certaines exécutions. Comme le nombre d'ordonnancements possibles explose lorsque le nombre de messages envoyés par l'application croît, des tests manuels ne peuvent explorer qu'une fraction des exécutions possibles. Nous utilisons la capacité du simulateur à contrôler l'exécution d'une application pour détecter automatiquement les deadlocks et les message races. Une première méthode pour réduire le nombre d'ordonnancements à tester utilise une réduction partielle d'ordre de l'espace de recherche et décompose l'exécution de l'application en sous-parties pouvant être testée indépendamment les unes des autres. Cette méthode s'appuie sur l'analyse statique de la trace d'une exécution de l'application et ne s'applique ainsi qu'aux applications produisant un ensemble fixe de messages, i.e. produisant les mêmes messages quel que soit l'ordre dans lequel ils sont traités par l'application. Nous supprimons cette limitation à l'aide d'une approche construisant dynamiquement un graphe d'état représentant les exécutions possibles. Les deux méthodes réduisent de plusieurs ordres de magnitude le coût d'un test, et leur combinaison améliore encore les résultats. Malgré tout, le temps nécessaire à tester un application reste prohibitif pour des applications plus compexes. Nous proposons donc aussi des algorithmes générant des sous-ensembles d'ordonnancements de messages révélant les erreurs de synchronization avec une probabilité élevée.

Ces dernière années, MPI s'est imposé comme le standard *de facto* pour écrire des applications parallèles à mémoire distribuée. La dernière partie de cette thèse décrit donc l'adaptation des techniques décrites ci-dessus pour tester des applications MPI. Nous décrivons d'abord l'extension de nos résultats sur la visualisation de l'exécution d'applications parallèles. Nous discutons ensuite les limites et les bénéfices liés à l'utilisation de graphes d'ordonnancements partiels d'exécutions pour décrire l'exécution d'applications MPI, et montrons comment des graphes d’états peuvent être construits dynamiquement.

Mots-clés : Applications parallèles à envoi de messages, prédiction de performance, message race, test, réduction partielle d’ordre, débogage, Dynamic Parallel Schedules, Message Passing Interface

Acknowledgments

I would first like to thank my thesis advisor, Roger Hersch for his support and guidance, and for letting me choose my way. Having the opportunity to lecture and teach various courses, describing the lab's Visible Human project to young children or presenting parallel programming to Master students was something I particularly enjoyed.

I am also greatly indebted to Sebastian Gerlach, and not only because he wrote the DPS framework on which most of this thesis is built. In addition to being a living encyclopedia of all things technical, Sebastian is also an entertaining and passionate communicator. Sharing an office for three years with him therefore widely broadened and deepened my knowledge on many subjects.

Somewhat further on the EPFL campus, I had the pleasure of meeting Ralf Gruber and Vincent Keller, who introduced me to the "practical" side of HPC. They were instrumental in making me realize the existing gap between a large part of the research being performed on parallel computing and the actual problems faced by the users. If this thesis tries to throw a few ropes between these two sides, it is thanks to them.

I also had a great time with the former and current members of the lab, Emin Gabrielyan, Fabienne Allaire, Mathieu Brichon, Thomas Bugnon, Romain Rossier, Fabrice Rousselle, Mathieu Hébert, Isaac Amidror, and Maria Anitua. Smart and friendly people also exist outside of the LSP, and I was privileged to meet a few of them. Julien Pilet, Roman Schmidt, David Leroux, Ron Levy and Jean Berney, thanks to all of you. A special thank goes of course to my friend, former flatmate and passionate researcher Ruben Merz, who was always ready for a late hour chat.

I had the pleasure of supervising a few semester and Master projects. In order of appearance: Pascal Jermini, Pierre Dumas, Noël Jobé, Selim Arsever, Samuel Robyr, Ali Al-Shabibi and Mamy Fetiarison. Thank you for your ideas, energy and contributions.

Finally, I would not have reached this point without the support of my family and, last but not least, of Ochélio. You changed my life for the better.

Contents

"Tell us, if there were one thing we could do for your village, what would it be?"

"With all respect, Sahib, you have little to teach us in strength and toughness. And we don't envy you your restless spirits. Perhaps we are happier than you? But we would like our children to go to school. Of all the things you have, learning is the one we most desire for our children."

– Conversation between Sir Edmund Hillary and Urkien Sherpa, from Schoolhouse in the Clouds.

Chapter 1

Introduction

1.1 Motivations

Parallel computers are spreading through our everyday life. This change has been mostly driven by the dropping costs of computer hardware. While the first parallel machines were expensive and produced by a few specialized companies, falling costs of commodity hardware made processing power affordable to smaller institutions and companies, and broadened the range of scientific disciplines that could take advantage of it. Some of the newcomers in the area of scientific computing such as life sciences and biology have become heavy users and now contribute to the stride towards more powerful computers.

More recently, processor manufacturers stopped aiming for ever higher clock rates. The performance increase is now mainly coming from adding multiple processing units within a single processor chip. These *multicore* processors are now present in nearly every new desktop and laptop computer. This shift has a strong impact on the program developers, who can no longer assume that their sequential programs will automatically run faster on the next hardware generation. Rather, clock frequencies are reduced to avoid heat dissipation issues caused by densely packed cores. The performance of individual cores therefore tends to decrease, and codes must be parallelized in order to take advantage of the additional processing units.

A third push toward a wider use of parallel processing may come from the raising prices of energy. The Green500 project (green500.org [102]) was specifically started to rank supercomputers taking their total electrical power consumption into account. The list ranks systems according to the number of floating-point operations performed per second for every unit of energy consumed (MFLOPS/Watt ratio [102]). It illustrates the benefits of swapping a few high-frequency processors by many lower frequency ones: among the most power efficient systems, IBM's BlueGene/P systems [52] contain densely packed processors clocked at only 850 MHz.

These developments imply that an increasing number of developers are being exposed to parallel programming. As with any technology, the average level of expertise of its users decreases as the popularity of the technology increases. It is therefore crucial to provide easy to use tools to help developers write efficient and correct parallel applications.

1.2 Parallel programming paradigms

Modern operating systems provide two levels of concurrent tasks execution: *processes* and *threads*. A *process* has access to a private address space, and cannot read or modify the address space of another process. This segregation is enforced by the operating system. Within a single process, a developer may use *threads* to perform concurrent computations. Multiple threads share a single address space[1]. These two concepts provide different parallelization paradigms.

In *shared memory*, *multithreaded* applications, a single process uses multiple threads. These threads share information by reading and writing common memory locations. This paradigm, once reserved for expensive multiprocessor machines such as the SGI Altix, is gaining in popularity due to multicore processors and to the availability of standardized APIs. POSIX threads [82] provide a portable interface for creating and managing threads on all major operating systems. OpenMP [12] is a more recent industry standard developed for facilitating the development of shared memory parallel scientific applications. Unlike POSIX threads, OpenMP frees the developer from managing the creation and destruction of threads, and provides easy to use and high-level synchronization functions. However, it mostly provides parallel for loops and therefore cannot implement arbitrary parallelization patterns.

In *distributed memory*, *message-passing* applications, multiple processes exchange data by sending messages to each other. The message-passing model has several advantages that explain its popularity for writing scientific applications. Firstly, multiple processes can run on a single multicore machine as well as on multiple machines. The same application can therefore run on a wider range of hardware. The same paradigm can be used whether messages are sent over a network or using shared memory. Secondly, message-passing applications enable the

[1]The terms of *heavy* and *light* threads are sometimes found in the literature. Some operating systems may also have more subtle mechanisms for sharing memory between processes. Nevertheless, these definitions are broadly accepted and are sufficient for our purpose.

developer to better control the distribution of the data among the participating processes. He may thus ensure that each piece of data is stored close to the process that uses it, which is beneficial to the application performance.

The current *de facto* standard for writing message-passing parallel applications is MPI (Message-Passing Interface [77]). MPI provides a portable interface for sending and receiving messages that abstracts the details of the underlying operating system and networking layer.

The shared and distributed memory parallel programming models are not mutually exclusive. For example, the availability of multicore processors makes the combination of OpenMP and MPI more common: MPI handles the inter-node parallelism, while OpenMP is used for intra-node parallelism.

The work presented in this thesis only considers distributed, message-passing parallel applications. Most of it relies on the Dynamic Parallel Schedules (DPS) parallelization library, which was developed at EPFL. Its distinguishing feature is the use of a *flow graph* to describe the application parallelism. This representation enables the DPS runtime to provide a number of features that would need to be implemented manually under MPI. The flow graph also provides a graphical representation that makes it easier to understand parallel programs and to communicate among developers.

1.3 Scope

While many problems can be parallelized easily in principle, naive parallelizations often have poor performance. The challenge is therefore to write parallel programs that efficiently use the available processing units. Among the factors that improve that efficiency are the choice of an appropriate problem decomposition granularity to minimize the amount of communications between processes, the overlap of the remaining communications with computations, and the reduction in the number of synchronizations between processes. Indeed, synchronizing a set of processes requires these processes to wait for the slowest process in the set. Synchronizations become increasingly prohibitive as the number of processes involved grows.

On the other hand, optimized implementations are complex and error prone. For instance, removing too many synchronizations may lead to errors that are specific to parallel applications: deadlocks occur when conflicts over the use of resources prevent one or several processes from moving forward, and message races occur when changes in the delivery order of messages change the computation outcome. Such errors are particularly difficult to foresee *a priori*, and to understand *a posteriori*. They are sometimes called *heisenbugs*: trying to study them changes the ordering of events in the application and makes the bugs disappear. Their transient nature, due to their appearance in specific and rare circumstances, adds an additional difficulty.

Application developers must therefore satisfy two objectives: produce parallel programs that are both efficient and correct. In this thesis, we present solutions for both sides of the problem. Simulation and performance prediction capabilities help identifying and setting the parameters leading to a good performance, and testing tools automatically detect potential synchronization errors.

One of the main considerations that guided the present work was the ease of use of the proposed solutions. In addition to the rising number of programmers exposed to parallel programming, many of the current computational science application developers do not have a computer science background. Rather, they are for instance chemists, physicists, biologists or mechanical engineers. All these users often prefer to focus on their main field of interest instead of spending time optimizing and testing code. For them, ease of use is a key element for the acceptance and use of tools facilitating the development of parallel applications.

1.4 Dissertation outline

We begin by describing the Dynamic Parallel Schedules parallelization framework. Chapter 2 presents the elements of DPS needed for understanding the subsequent chapters, and describes a few extensions implemented during the course of this thesis.

Chapter 3 then discusses the integration of simulation capabilities within the DPS framework, which can be used for any DPS application. We propose a partial direct execution technique that prevents the need to use a parallel machine to run simulations and at the same time reduces the running time and memory consumption of simulations.

Testing the application for the absence of deadlocks and message races requires executing an application in many different configurations. Unfortunately, the number of possible executions explodes as the size and complexity of the application grows. In Chapter 4, we describe methods that greatly reduce the number of tests needed to exhaustively cover all possible executions and detect existing errors. For cases where the testing costs remain prohibitive, we present heuristics that test a subset of executions that are likely to produce errors.

The following chapter, Chapter 5, briefly describes the Message Passing Interface (MPI) standard, which is the most popular API for writing distributed memory parallel applications. We then show how the testing techniques developed in the context of DPS in Chapter 4 can be adapted to MPI applications.

Finally, Chapter 6 summarizes our results and the possible paths for future research.

1.5 Contributions

The detailed list of contributions is as follows:

- We present an efficient simulation framework for predicting the performance of adaptive parallel applications.
- We describe a simple parameterization of cluster hardware that enables accurate prediction of parallel application running times on clusters of single-processor nodes.
- For parallel applications that can be described using a static Partial-Order Execution Graph, we propose an efficient decomposition algorithm for reducing the number of ordering to be tested for detecting races within DPS applications.
- We describe a dynamic approach for testing parallel applications whose dependencies cannot be statically expressed.
- These two approaches provide a significant reduction in number of orderings to be tested. However, their scalability is limited with respect to the increase in the number of nodes and in the communication complexity. We therefore describe heuristics that test a subset of possible orderings that have a high probability of producing existing message races.
- We generalize the use of the Partial-Order Execution Graph and of the dynamic testing approach to detect message races within MPI parallel applications.
- For both DPS and MPI applications, we describe debuggers that provide a graphical representation of the execution. In both cases, the tools have the ability to reorder events in order to test for the presence of message races.

Chapter 2

The Dynamic Parallel Schedules Framework

2.1 Introduction

The Dynamic Parallel Schedules (DPS) framework is a high-level framework for developing distributed memory parallel applications on clusters of workstations. It has been developed at the Peripheral Systems Laboratory of the Ecole Polytechnique Fédérale de Lausanne (EPFL). While its general concepts were developed in the late 90's, its implementation reached its final form in 2005. The various incarnations of DPS have been used to teach parallel programming to Master students, who have successfully compiled and run their parallel applications on a wide range of compilers, operating systems and hardware platforms. DPS therefore provides a stable and cross-platform library for writing parallel applications.

An application using DPS is expressed as a directed acyclic graph of sequential operations, called a *flow graph*. Individual operations are fully customizable, and sections of distinct flow graphs can be composed together, enabling the reuse of existing code. The graphs and the mapping of operations to processing nodes are specified dynamically at runtime. DPS applications are pipelined and multithreaded by construction, ensuring a maximal overlap of computations and communications.

This chapter presents the concepts and features of DPS that are required to fully understand the topics presented in the following chapters. In particular, it leaves out many practical implementation details about how to write DPS applications. This information can be found on the DPS website [37], which provides tutorials, sample code and the API documentation. We also do not detail the internal implementation of the framework, and ignore several advanced features offered by DPS. These can be read in [35].

2.2 The flow graph

DPS applications are described by their *flow graph*. The nodes on the graph are the operations that are executed. Its edges specify how messages may flow from one operation to the next, thereby describing the admissible sequences of operations. Each operation in the graph takes one or several message as input, and produces one or several messages as output.

The flow graph only specifies the dependencies between the various operations that compose an application, and not the actual application deployment. The allocation of operations onto processing nodes will be described in Section 2.8.

2.2.1 Basic Operation Types

DPS provides four basic operation types, which differ in the number of messages that they may receive or send. The basic operations are the following:

- *Split* operations take exactly one message as input, and can produce any number of messages as output. At least one message must be sent to avoid halting the data flow in the flow graph. Outputs typically represent subtasks that may be performed in parallel.
- *Leaf* operations take exactly one message as input, and produce exactly one message as output. They are typically used to perform computations using data provided in the input message, and send the results within the output message.
- *Merge* operations take any number of messages as input, and produce exactly one message as output once all messages have been received. These operations are used to merge partial results into a single result. In a flow graph, every merge operation must match a single split operation. The number of input messages of a merge corresponds to the number of messages sent by the matching split operation.
- *Stream* operations combine the functionalities of a merge and a split operation by allowing any number of both input and output messages. Every stream must therefore match a split or another stream operation that precedes it in the flow graph, as well as a successor merge or stream operation.

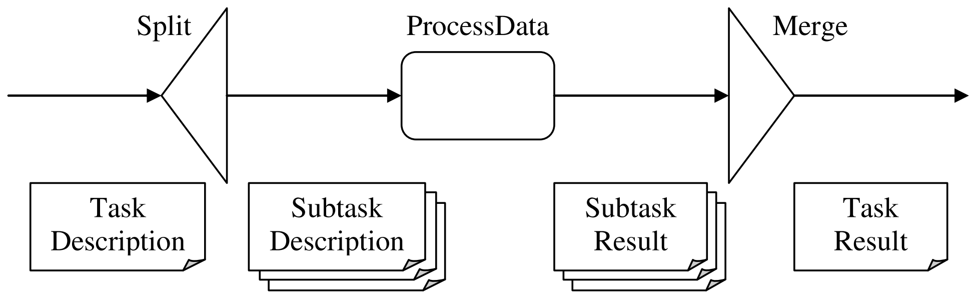


Figure 2.1: *A simple parallel application using a split-leaf-merge flow graph.*

In order to build a parallel application, a developer writes custom operations of the desired type and composes them into a flow graph. Figure 2.1 displays the basic building block of parallel applications written using DPS. The split-leaf-merge construct forms a simple flow graph suitable for many types of embarrassingly parallel applications. The split and merge operations contain customized code to control exactly how the work and the data is distributed in subtasks, and how the computed sub-results are combined into the final output result. The code of the leaf operation processes the output messages of the split operation and sends a message to the merge operation.

The basic split-leaf-merge construct may have multiple parallel branches when different outputs of a single split or stream operation are processed by different operations, as shown in Figure 2.2.

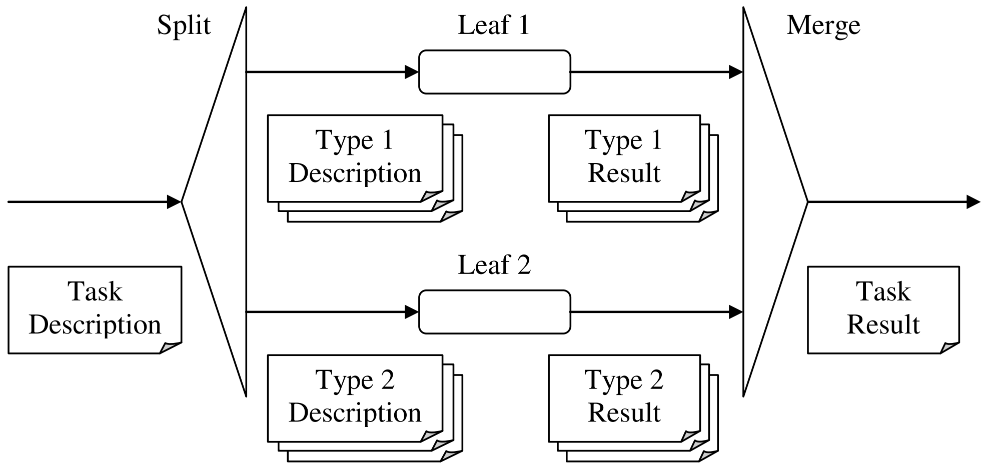


Figure 2.2: *Parallel branches enclosed by a split and a merge operation pair. The two branches perform different functions.*

2.2.2 Flow Graph Composition

Operations may be composed together provided that every split operation is matched by a single merge operation. Since streams combine the functionality of both splits and merges, the left side of a stream must match a split or another stream, and its right side must match a merge or another stream.

From the perspective of the flow graph, a sequence of two leaf operations is equivalent to a single leaf operation, since for each input message, there is exactly one output message. This equivalence property also applies to the simple split-leaf-merge block: for every input of the split operation, the matching merge produces exactly one output message.

Admissible flow graphs may therefore be built by substituting leaf operations by sequences of leafs or by split-leaf-merge blocks. Starting from a single leaf operation, one may produce flow graphs of arbitrary complexity by iteratively performing operation substitutions (Figure 2.3).

Multiple computations steps can be performed by using consecutive split-leaf-merge sequences, as illustrated in Figure 2.4a. Since the merge operation does not send any message before all incoming messages have been received, it guarantees that all previous computations are over before starting the next operation in the flow graph. In some cases however, this synchronization model is too strict and negatively impacts the application performance. This happens when not all the results of the computations preceding the merge are required in order to perform parts of subsequent computations. In these cases, it is desirable to already split out these parts of the second computation before the first computation is complete in order to ensure proper pipelining of the application. The stream operation was designed to accomplish this objective, by combining the functionality of a merge operation and a split operation in a single operation. The stream operation can output data objects at any time within the merge process. From the perspective of the flow graph, any merge-split sequence can be replaced by a stream operation, as illustrated in Figure 2.4b.

2.2.3 Loops

The actual number of messages sent by a split or stream operation depends on the implementation of the operation, and is therefore known only during the execution of the flow graph. However, the flow graph itself cannot be modified once its execution started. Although flow graphs of arbitrary lengths can be created statically, in many cases applications do not know in advance how many operations will have to be executed. For instance, iterative linear solvers

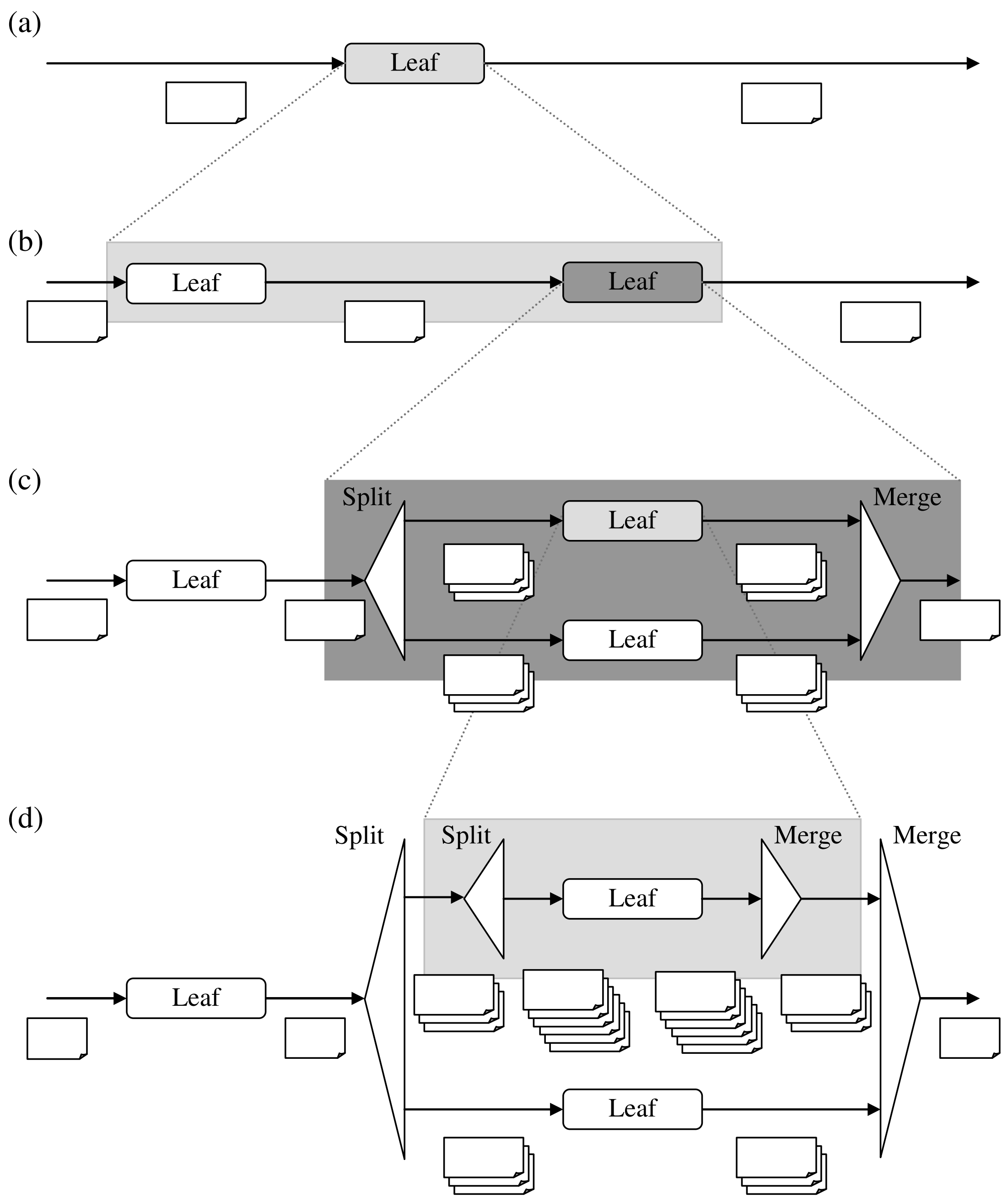


Figure 2.3: *Construction of a complex flow graph by subsequent substitutions of leaf operations.*

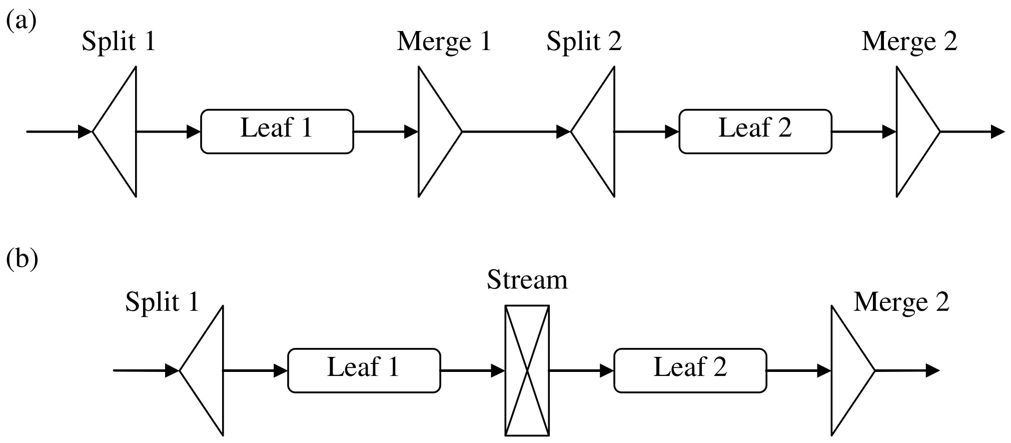


Figure 2.4: *(a) Two computations with intermediate barrier synchronization, and (b) relaxed synchronization using a stream operation*

iteratively compute an approximate solution x' of a linear system $Ax = b$. Each iteration improves the approximation of the solution and the execution stops once the approximation error $||Ax' - b||$ is smaller than some predefined ϵ. In such cases, the number of iterations to be performed cannot be determined a priori given A.

Within a flow graph, this type of execution pattern can be expressed using a loop, a specialized type of operation that encloses any sequence of operations and evaluates a condition on the output message of the last operation of the sequence. As long as the condition is true, the encapsulated sequence of operations is executed again on the output message.

The loop operation does not create a cycle within the flow graph; it produces a pipelined sequence of operations, the length of which is determined at runtime based on the loop condition. Flow graphs are therefore always acyclic, since none of the allowed patterns enables the construction of cycles. The acyclic property ensures that unless an operation does not generate any output message an execution is always free of deadlocks.

Figure 2.5 illustrates some flow graph constructions that can be achieved using loop constructs. When inserting loop constructs, the only constraint is to ensure that the flow graph preserves the symmetry between split and merge operations: for each split operation within the flow graph, a corresponding merge operation needs to exist within the flow graph.

2.3 Threads and thread collections

Flow graphs provide a simple and efficient mechanism for describing program flow. We now need to indicate in which context the various operations on the flow graph should be executed.

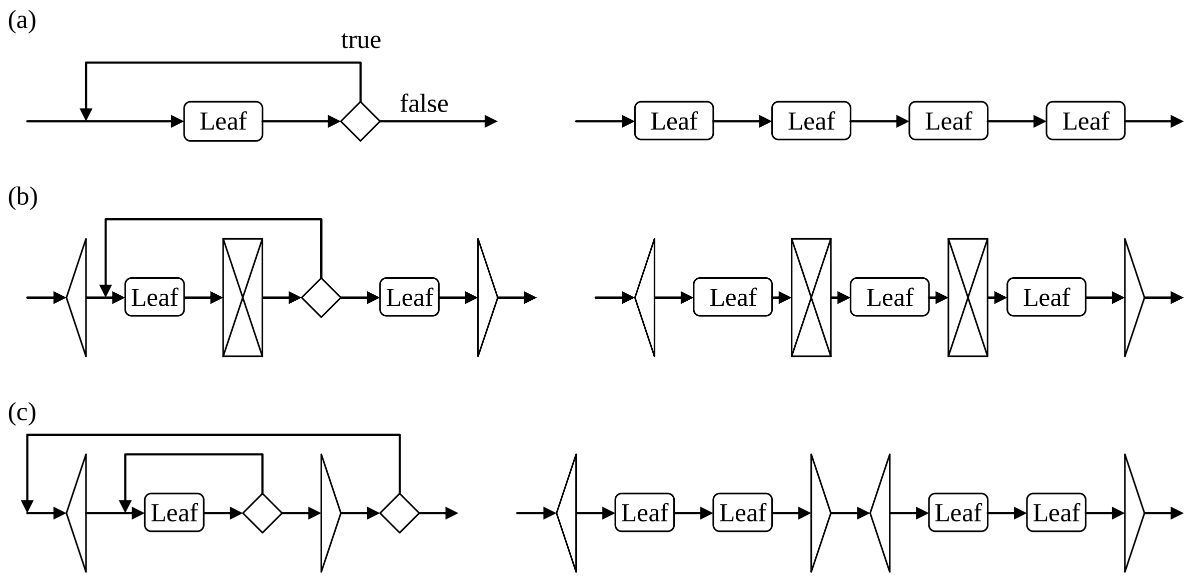


Figure 2.5: *Left: examples of use of the loop construct; right: resulting runtime flow graph. (a) Loop around a leaf operation, returns false at fourth iteration; (b) deep pipeline using stream operations, loop returns false at second iteration; (c) nested loops, both return false at second iteration.*

Operations are assigned to *threads* that are mapped onto the processing nodes. A thread within the parallel schedules concept provides a context within which operations can be executed. A thread executes only one operation at a time. The threads are grouped within *thread collections*. Multiple threads (whether from the same or from different thread collections) may be mapped onto the same compute node, in which case they execute operations concurrently and independently of each other.

Each operation within the flow graph must be attached to one thread collection. A single thread collection may support multiple operations. Let us for instance consider a simple compute farm application, where a master node distributes tasks to a set of processing nodes, and later collects the results of the processing. For such an application, two thread collections would be created. The first is used for all the master tasks, and we attach the split and merge operations to it. The second thread collection is used for all the processing tasks, and runs the leaf operations. Figure 2.6 displays an example where the same operation is executed twice in a pipeline, and each operation runs on a different collection of threads. In this example, the master thread collection would contain only a single thread, whereas processing thread collections would typically contain one thread for each participating CPU.

The thread collections provide a logical description of the execution environment of the application. Their assignement to operations is invariant for a given flow graph. On the other hand, the actual number of threads within a thread collection, as well as the their distribution

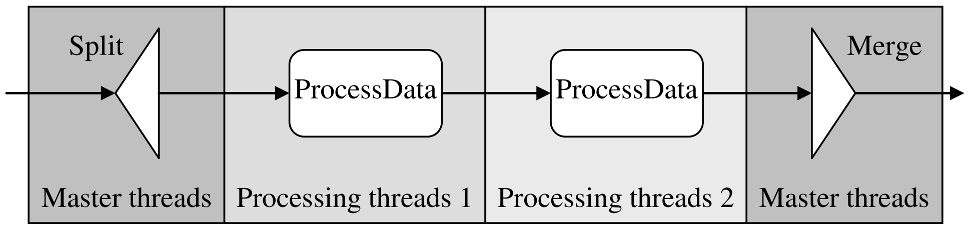


Figure 2.6: *Assignment of thread collections to operations within the flow graph. The ProcessData operation is executed twice in a pipeline on two different thread collections.*

onto compute nodes, can be chosen arbitrarily at runtime and may potentially change during the parallel schedule execution (Section 2.8.1).

2.3.1 Thread local storage

Parallel schedule threads can provide local storage to the operations that execute within their context. The storage is provided as an instance of a user-defined data structure. The *type* of storage is the same for all threads of a same thread collection. However, the *storage* itself is specific to individual threads, and two threads of the same type located on the same processing node cannot access each other's data.

This local storage is preserved within the thread state, and persists from one operation to the next. Data-parallel applications can use this thread local storage to store distributed data structures. For example, an application performing matrix computations could store matrix blocks within its threads.

Since local storage provides a private memory location to each thread, the programming model of DPS can be seen as being strictly distributed memory, even though multiple operations may be executing simultaneously in the same process[1].

2.3.2 Routing functions

When an operation outputs a message, the DPS runtime identifies the recipient operation and therefore the destination thread collection using information from the flow graph. However, it must still determine which thread within the collection should be used as the execution context for running the operation.

In the data-driven computation model of DPS, the destination thread of each message is

[1]Developers may technically use global variables to share information between threads. This practice is however discouraged, among other reasons because it prevents threads from migrating between processes (Section 2.8.1).

computed using a routing function. Routing functions are attached to operations during the construction of the flow graph, and are applied to each one of their input messages. Since a routing function may access the content of messages to determine their destination, its behavior is very flexible: it can provide simple mechanisms such as constant routing or round-robin routing, data-dependent routing using the message content, or even automatic load-balancing (Section 2.6.3).

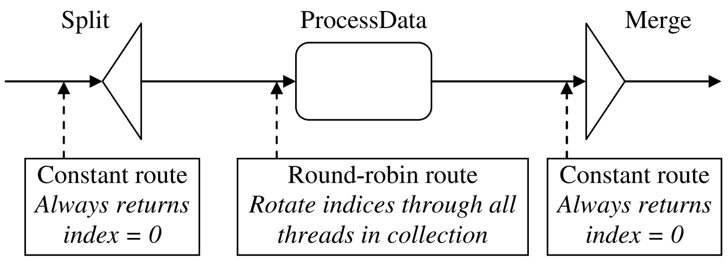

Split
ProcessData
Merge
Constant route
Always returns index = 0
Round-robin route
Rotate indices through all threads in collection
Constant route
Always returns index = 0

Figure 2.7: *Routing functions are attached to operations, and are used to compute the destination thread of incoming messages.*

2.4 A practical example

The previous sections described the basic concepts of DPS. We now present the implementation of a simple parallel application so as to provide a better understanding of the behavior of DPS applications. This section illustrates only the most important elements presented so far, and we refer the interested reader to the documentation provided on the DPS website [37] for further information.

All the elements that compose flow graphs such as operations or routing functions are implemented as C++ classes. These statically defined components are assembled at any time during execution to build one or several flow graphs. A same component may be used multiple times within a single flow graphs and can be reused in multiple flow graphs. A same flow graph can then be used to spawn multiple parallel schedules.

The chosen example is a basic parallel merge sort application. The parallelization is expressed using a split-leaf-merge flow graph similar to the one displayed in Figure 2.1. First, the *SplitVector* operation receives a vector of integers as input, and splits it into subvectors containing 100 elements. The leaf operation, *Sort*, then receives a subvector and sorts it. The *MergeVector* then aggregates all the sorted subvectors into a single sorted vector. The implementation focuses on brevity rather than on performance, enabling the complete source code to be provided.

2.4.1 Messages

The messages transiting through the flow graph are also C++ objects. Our merge sort application uses only a single type of message. The message contains two fields, a vector provided by the C++ standard library that contains the data to be sorted, and an integer to be used by the routing function to determine the destination thread of the message.

The C++ language provides no reflection and serialization capabilities. DPS therefore provides its own mechanism that is able to automatically serialize regular C++ objects. It requires only a few macros to be added by the application developer, and will be described in more details in Section 2.7. Listing 2.1 displays the declaration of a serializable *VectorData* class that contains the required members: one integer, *target*, to be used by the routing function and one standard library vector, *v*, to store the data.

Listing 2.1: *Serializable object declaration*

```
#include <dps/dps.h>

class VectorData : public dps::AutoSerial
{
  CLASSDEF(VectorData)            // Class name
  MEMBERS
    ITEM(int, target)             // int target;
    ITEM(std::vector<int>, v)     // std::vector<int> v;
  CLASSEND;

public:
  // Default constructor
  VectorData() { target = 0; }
};
```

2.4.2 Operations

Operations constitute the nodes of the flow graph, and encapsulate all of the application's functionality. Within DPS, all operation types (leaf, split, merge and stream) share a common syntax and programming model. The body of an operation is composed of standard sequential C++ code that performs the operation's tasks. Operations are further characterized by two additional parameters specifying the type of input messages and the type of output messages.

Listing 2.2 shows the code of the *SplitVector* operation. The type (split, merge, leaf, stream) of each operation is determined by its base class. The *SplitVector* class derives from the *dps::SplitOperation* class, and specifies the types of the input and output messages as tem-

Listing 2.2: *Declaration of* SplitVector *operation*

```
15 // Splits input vector into 100 element subvectors
16 class SplitVector
17   : public dps::SplitOperation<VectorData, VectorData>
18 {
19   IDENTIFY(SplitVector)
20 public:
21   void execute(VectorData *in)
22   {
23     int counter = 0;
24     std::vector<int>::const_iterator it = in->v.begin();
25     while(it != in->v.end())
26     {
27       VectorData *out = new VectorData();
28       out->target=counter;
29       for(int i=0; i<100 && it!=in->v.end(); ++i)
30         out->v.push_back(*it++);
31       postDataObject(out);
32       counter++;
33     }
34   }
35 };
```

plate parameters[2] (line 17). An *IDENTIFY* macro adds code that registers the type of the object within a *class factory* in the DPS library. When a message arrives, the DPS runtime is therefore able to instantiate the operation that will process the message.

The entry point of an operation is its *execute* method, which takes the received message as a parameter. The developer must provide his own code to process the incoming message and create and send new messages. In our example, the *SplitVector::execute* method creates new messages containing 100 elements from the original vector. The local variable *counter* keeps track of the number of messages already created and is used to set the *target* field (line 28), thereby numbering every message. Once a message is instantiated and initialized, it is sent to the next operation using the *postDataObject* method. Once all messages have been sent, the *execute* method returns and the operation terminates.

The leaf operation *Sort* is created in the same fashion (Listing 2.3). It derives from the *dps::LeafOperation* base class and specifies *VectorData* as the type of its input and output messages (line 38). Its *execute* method first creates a copy of the input message. It then sorts the vector of the message copy, and sends it.

The *MergeVector* operation (Listing 2.4) derives from the *dps::MergeOperation* base class.

[2]Multiple input and output types can be specified using type vectors [37].

Listing 2.3: *Declaration of* Sort *operation*

```
// Sort subvector
class Sort
  : public dps::LeafOperation<VectorData, VectorData>
{
  IDENTIFY(Sort);
public:
  void execute(VectorData *in)
  {
    VectorData *vd = new VectorData(*in);  // Copy message
    std::sort(vd->v.begin(), vd->v.end()); // Sort vector
    postDataObject(vd);                    // Send result
  }
};
```

Listing 2.4: *Declaration of* MergeVector *operation*

```
// Merge parts back
class MergeVector
  : public dps::MergeOperation<VectorData, VectorData>
{
  IDENTIFY(MergeVector)
public:
  void execute(VectorData *in)
  {
    VectorData *out = new VectorData();
    do
    {
      std::vector<int> tmp;
      tmp.swap(out->v);   // Exchange content of tmp and out->v
      std::vector<int>::const_iterator it1 = in->v.begin(),
                                       it2 = tmp.begin();
      while (it1!=in->v.end() || it2!=tmp.end())
      {
        if (((*it1)<(*it2) && (it1!=in->v.end()))
            || it2==tmp.end())
          out->v.push_back(*it1++);
        else
          out->v.push_back(*it2++);
      }
    }
    while((in=(VectorData*)waitForNextDataObject()) != NULL);

    postDataObject(out);
  }
};
```

The operation starts by creating the output message into which sorted subvectors will be merged. While the first message is received as a parameter of the *execute* operation, subsequent messages are received by calling the *waitForNextDataObject* method (line 73). Calling this method suspends the execution of the operation until the next message arrives. For each input message, we create a temporary *tmp* vector to store the elements already accumulated in the output message (line 61; following the swap, *out->v* is empty and *tmp* contains the values previously in *out->v*). The content of *tmp* and of the input message *in* are then merged into the output message. When all messages have been received, *waitForNextDataObject* returns *NULL*, and the operation sends its output message (line 75).

2.4.3 Routing function

The routing functions used to determine the destination thread of messages are also defined as C++ classes. They derive from the *dps::Route* base class, and specify the type of routed messages as a template parameter. The message to be routed is delivered as a parameter of the *route* member function, which is responsible for computing the index of the destination thread of the message. The body of that function may do so using any computations, possibly using member variables of the routed message. In our case, we use the *target* field modulo the number of threads in the thread collection to which the next operation is attached (Listing 2.5).

Listing 2.5: *Routing function declaration*

```
78 // Routing function
79 class TargetRoute : public dps::Route<VectorData>
80 {
81   IDENTIFY(TargetRoute)
82 public:
83   Size route(VectorData *in)
84   {
85     return in->target%threadCount();
86   }
87 };
```

2.4.4 Building and running the flow graph

The building blocks of our merge sort application have now been declared. In order to build a flow graph and execute a parallel schedule, we must define an application class that derives from *dps::Application*. The main method within that class is *start*, and parallel schedules must be started from within that method.

Listing 2.6: *Build and execute flow graph*

```cpp
//! Application class
class MergeSortApp : public dps::Application
{
  IDENTIFY(MergeSortApp)
public:
  //! Startup function
  virtual void start()
  {
    // Create thread collections
    dps::StatelessThreadCollection mainThreads =
      getController()->createStatelessThreadCollection("main");
    dps::StatelessThreadCollection processThreads =
      getController()->createStatelessThreadCollection("process");

    // Create threads on processing nodes
    mainThreads.addThread("host1");
    processThreads.addThread("host1 host2 host3");

    // Declare flow graph nodes with operation, routing function,
    // and thread collection
    dps::FlowgraphNode<SplitVector, TargetRoute> split(mainThreads);
    dps::FlowgraphNode<Sort, TargetRoute> sort(processThreads);
    dps::FlowgraphNode<MergeVector, TargetRoute> merge(mainThreads);

    // Build flow graph
    dps::FlowgraphBuilder builder = split >> sort >> merge;
    dps::Flowgraph sortGraph =
      getController()->createFlowgraph("mergeSortGraph", builder);

    // Create initial message with 10000 random elements
    VectorData *in = new VectorData();
    for(int i=0;i<10000;++i)
      in->v.push_back(rand()%10000);

    // Execute parallel schedule
    VectorData *result =
      (VectorData*)getController()->callSchedule(sortGraph, in);

    // 'result' is the output message of the MergeVector
    // operation and contains the sorted vector

    delete result;
  }
};
```

We first create the thread collections onto which the flow graph will execute. In our case, we create stateless thread collections, i.e. collections where threads have no attached local storage (lines 97–100). We then add threads to each collection using their *addThread* member function. The parameter is a string that typically contains the hostnames or IP addresses of the processing nodes onto which the application will execute.

The flow graph is built by combining *flow graph nodes*. Constructing a flow graph node requires three parameters. The first two specify the type of the operation and the type of the routing function used to route the input messages of the operation, and appear as template parameters. The thread collection providing the execution context to the operation is specified as a constructor parameter (lines 108–110). DPS performs type validation in order to ensure that the operations will be executed on threads of the appropriate type, and that the routing functions use the correct message type in order to select the target threads. However, operations that do not perform any processing on a locally stored thread state can be attached to a thread collection containing threads of any type.

The flow graph nodes may then be connected to each other to form a flow graph (line 113). The resulting sequence of operations is validated at compile time to ensure that the input type of an operation matches the output type of the preceding operation. Once the flow graph has been built, a parallel schedule can be started with the *callSchedule* method. The two parameters indicate the flow graph to be started, as well as its input message, i.e. the message that will be delivered to the first operation of the flow graph (line 124).

Finally, one must instantiate and start the application itself. The application object is instantiated explicitly by the developer and may therefore be set up as needed. The *dps::dpsMain* function then takes care of initializing internal data structures of the DPS runtime, and calls the application *start* method. This is done in Listing 2.7 at line 135.

Listing 2.7: *Start application*

```
132 //! Starts up application
133 int main(int argc, char *argv[])
134 {
135   return dps::dpsMain(argc, argv, new MergeSortApp());
136 }
```

2.4.5 Using local thread storage

Local thread states are expressed as standard C++ classes, and may therefore contain any number of fields to store data. While operations are destroyed after execution, threads are created once. They therefore preserve their variables, which can be used by multiple operations that run on the same thread. Such variables may be initialized by adding a constructor without any parameters to the thread state class. This constructor is called when the thread is created by DPS. Within an operation, the local storage can then be accessed by calling the *getThread* member function.

Let us assume that the *Sort* operation wants to store the sorted vector for future use. One would first need to declare a class able to store the vector (Listing 2.8). We must then add a template parameter to the *Sort* operation that indicates the type of thread it has access to (line 9). The operation may then access the local thread state via the *getThread* method (line 18). The type of the thread storage class must finally be added as a template parameter during the

Listing 2.8: *Declaration of local thread storage and access from operation*

```
 1 // Declare VectorThread thread storage
 2 class VectorThread
 3 {
 4 public:  std::vector<int> threadVector;
 5 };

 7 // Sort subvector and store it within local thread state
 8 class Sort
 9   : public dps::LeafOperation<VectorData,VectorData,VectorThread>
10 {
11   IDENTIFY(Sort);
12 public:
13   void execute(VectorData *in)
14   {
15     VectorData *vd = new VectorData(*in);
16     std::sort(vd->v.begin(), vd->v.end());
17     // Store sorted vector locally
18     getThread()->threadVector = vd->v;
19     postDataObject(vd);
20   }
21 };


24 // In the start() method of the application class
25 dps::ThreadCollection<VectorThread> processThreads =
26   getController()->createThreadCollection<VectorThread>("process");
```

declaration of the thread collection within the *start* method of the application: this tells DPS which storage must be created upon creation of the threads of the collection.

2.4.6 Execution highlights

The execution of DPS application can be represented graphically by drawing its *unfolded* flow graph. Such a view better represents the parallelism within the computation by displaying the distribution of operations onto threads. However, in order to keep the representation compact, it does not tell how many times a single operation is executed on a given thread (Figure 2.8).

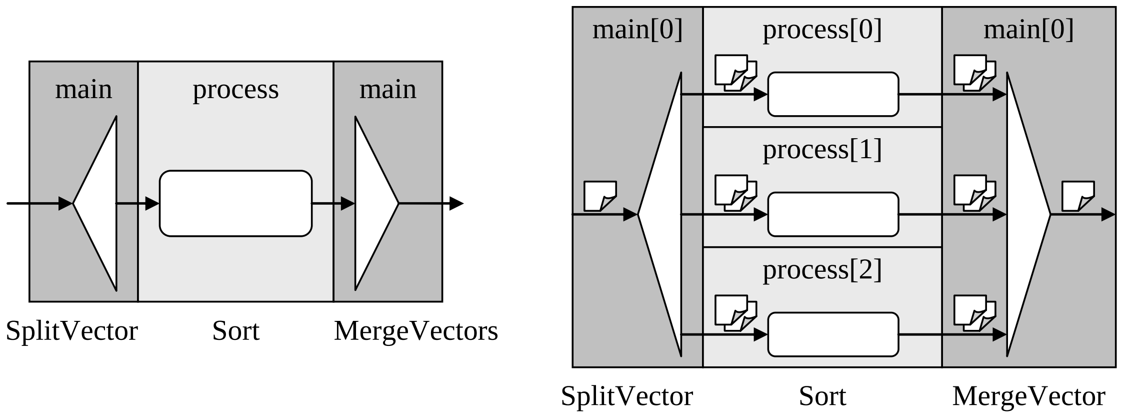


Figure 2.8: *(a) The flow graph computing the parallel merge sort and (b) its unfolded representation when the processing thread collection contains three threads. Although DPS instantiates one* Sort *operation for every output message of* SplitVector*, we only display a single operation per thread to keep the graph readable.*

After the call to the *callSchedule* method, the input message of the flow graph first goes through *TargetRoute*, which is the routing function associated to the first operation of the flow graph. Since the value of *target* is 0 (as set by the default constructor of *VectorData*), the message is routed to the thread 0 of the thread collection “main”. Upon reception of the message, the thread instantiates a new *SplitVector* operation and calls its *execute* method with the received message as parameter. All the output messages of the *SplitVector* operation are then routed by *TargetRoute*. Since the value of *target* in successive messages are consecutive integers, the routing functions effectively distributes the messages to the threads of the “process” thread collection in a round-robin fashion. The outputs of the *Sort* operations are then routed back to thread 0 of the “main” thread collection where they are collected by the *MergeVector* operation. (The thread collection “main” contains a single thread, so *TargetRoute::route* always returns 0). Since the merge operation is the last operation in the flow graph, its output message is sent back to the caller of the parallel schedule. The result is then available for further processing within the *start* method of the application.

2.5 Runtime behavior of parallel schedules

DPS implements a data-driven execution model, where all communications between DPS operations are performed asynchronously. Each thread stores incoming messages within a queue until it can process them. As soon as a message is available within a thread's pending message queue, that thread immediately starts processing by executing the appropriate operation. When the execution of the operation completes, the next message within the queue is retrieved for processing. Similarly, messages are sent to the target thread as soon as they are created by an operation. This behavior allows the overlapping of computations and communications and hides at least partially the latencies and transfer times related to communications.

2.5.1 Message identifiers

Every message sent is wrapped within a *token*. Tokens contain additional fields that are used internally by DPS to identify the address of the destination process, the index of the destination thread, and the identifier of the flow graph node that will process the message. This information is used by the receiving process to pass the message to the appropriate operation. In the context of this thesis however, the most important field is the message identifier.

A first component of message identifiers is a list of integers. For leaf operations, that list is identical in the input and output messages. For split operations, the list of the output token contains all the elements of the list of the input token, plus an additional integer indicating the number of messages that have already been posted by this split. The additional integer ensures that all messages posted by a given split operation are uniquely identified, yet share a common prefix. For merge operations, the list attached to the output token is a copy of one of the input token without the last element. Therefore the list of integers of the output token of a merge operation is identical to the list of the input token of the corresponding split operation. An example is illustrated in Figure 2.9. Since stream operations behave like a combination of a split and a merge operation, they have the same message identifier length on their input and output. The last element of the output token identifier follows the counting pattern seen in split operations.

The message identifier is used to control the instantiation of new operations when messages need to be processed. Split and leaf operations are always instantiated, since they only process

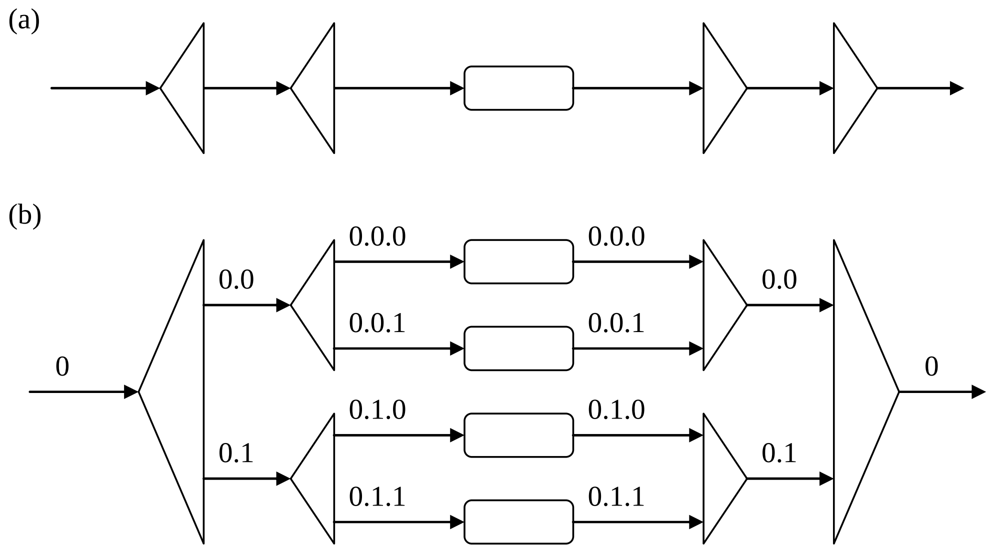


Figure 2.9: *(a) A two-level split flow graph and (b) the integer lists attached to each message when each split sends two messages.*

a single input message. Merge and stream operations however need to process multiple input messages within the same context. This context is identified by examining the token identifier of incoming messages of merge and stream operations. Since split operations append one unique element to the identifier of all the messages that they post, all the messages that share a common identifier with the exception of the last element are processed by the same merge or stream operation.

Simple hierarchical numeric identifiers are sufficient in simple flow graphs without loops. However, when loops are introduced, messages pass through the same flow graph parts multiple times. Since within the loop the operations are replicated, and since operations created in different loop iterations may be active at the same time, an additional factor needs to be added to the message identifier in order to distinguish the different instances. This is achieved via a counter, which counts the number of operations that have been performed on a message at a given level of the hierarchy. Since the operation counter is incremented at least once between every pass of a loop, it ensures that the resulting message identifiers are unique even when the same flow graph node is reused.

Figure 2.10 illustrates two simple flow graphs using loops. The first loop contains a split-merge pair, whereas the second loop contains a stream operation. The corresponding unfolded parallel schedule runs are also shown, together with the message identifiers circulating along the graph edges. The stream operation has exactly the same effect on the counter as a split-merge operation pair.

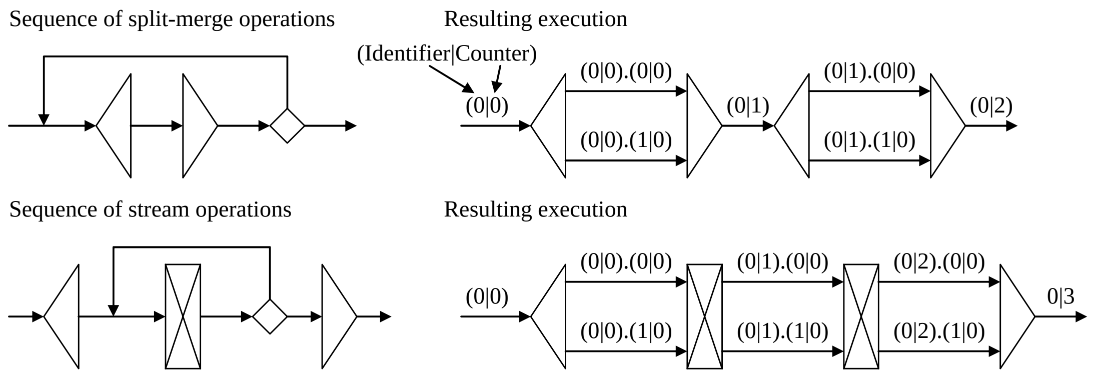


Figure 2.10: *Loops and resulting parallel schedule execution (operations have been omitted for simplicity)*

The aforementioned scheme guarantees the uniqueness of message identifiers within a flow graph execution. However, it does not guarantee that messages always have the same identifier in different executions. For operations with multiple output messages, the last integer of the list is set using a counter. For stream operations, where the ordering of output messages may depend on the ordering of input messages, this implies that the same message may have distinct identifiers in different executions of the stream operation. For this reason, the *postDataObject* method may take one additional integer parameter that is substituted to the message counter value. It is therefore up to the developer to set this counter in a way that is specific to each message. This completes the ability of message to be uniquely and deterministically identified by the runtime system.

This determinism of message identifiers was first required for implementing the fault-tolerance mechanism of DPS [35, 36, 38]. When a process crashes, some of its operations may have to be reexecuted, which produces new messages. If the destination thread of these messages did not crash, it may have received some of these messages already after the first execution of the operations. The DPS runtime therefore uses message identifiers to detect duplicate messages, assuming that a message will always have the same identifier. In this thesis, the uniqueness and determinism properties of message identifiers will be critical for implementing the message race detection techniques in Chapter 4.

2.5.2 Execution model

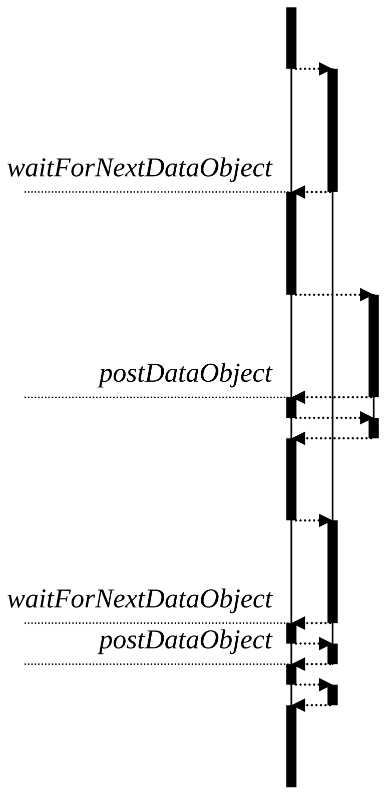
All operations within parallel schedules are executed in the context of a DPS thread, which owns one operating system thread and a queue of tokens that need to be processed within its context. Each thread then simply takes the first token out of its queue, reads the message identifier to discover which operation needs to be executed and instantiates a new operation if necessary. It finally calls the operation's *execute* method with the enclosed message. User messages are processed on a first in, first out basis.

Since DPS threads run on dedicated operating system threads, they run asynchronously, interacting with the rest of the system only through their message queue. However, there are cases where an operation cannot continue execution within the current context. The first is when a merge operation calls the *waitForNextDataObject* method: the merge blocks until the next message is available or the number of message sent by the corresponding split operation is reached. The second occurs when a split operation that reached its flow control limit calls *postDataObject* to send another message (flow control will be described in Section 2.6.2). Since they combine the functionality of split and merge operations, stream operations may be suspended in both cases.

This ability to suspend split, merge and stream operations is essential to execute applications asynchronously without deadlocks. For instance, since there is no constraint on which operations may share a same thread, multiple leaf operation may run on the same thread as their successor merge operation. Without operation suspension, starting the merge operation before all leafs have executed would cause a deadlock. Indeed, the merge would wait for incoming messages while holding the thread necessary to run the operations that may produce these messages.

Calls to *waitForNextDataObject* and *postDataObject* are performed in user-written code within the operation's *execute()* method. In order to suspend an operation, the DPS thread therefore needs to maintain the complete execution context (i.e. the stack frame) of the operation until it may be resumed [113]. The DPS library creates one stack frame for each executing operation[3]. When an operation is suspended, it is stored within the DPS thread it is associated with. DPS switches back to the thread's main stack frame, which is then able to process other incoming messages. When it receives a message destined to a suspended operation, the thread switches back to the operation's stack frame to resume it from the same point. Figure 2.11 shows the stack switching in effect for a thread that processes three messages, two of which are processed by the same merge operation. Stack frame switching provides an inexpensive method for maintaining multiple streams of execution simultaneously without the overheads of using multiple threads.

[3]Functions managing stack frames are provided by the operating systems (*fibers* in Windows, *ucontext* in UNIX variants).



Initial state of the thread, running in its own stack frame

A message arrives; the thread creates the corresponding merge operation with its own separate stack frame. Execution switches to the merge operation stack frame. When the merge operation completes processing of the message, it calls *waitForNextDataObject*. Execution switches back to the thread stack frame, and the merge operation is suspended.

Another message arrives; the thread creates the corresponding leaf operation with its own stack frame. Execution switches to the leaf operation stack frame until the leaf operation calls *postDataObject*. Execution switches back to the thread stack frame in order to process the sent message, and returns immediately to the leaf operation stack frame. The leaf operation terminates and control returns to the thread stack frame.

Another message arrives; the thread returns control to the corresponding merge operation. Execution switches to the merge operation stack frame. The merge operation processes the message and calls *waitForNextDataObject* again.

This was the final message for the merge operation. The executor returns NULL to *waitForNextDataObject*, the merge operation sends its output message, passing control back to the executor, who returns it immediately to the merge operation. The merge operation terminates and returns control to the thread stack frame.

Figure 2.11: *Stack frame management in DPS threads*

2.6 Split-merge interactions

The application developer does not need to specify how many messages are to be received by a merge operation. Merge operations are able to automatically determine the expected number of messages, and pass that information to the developer by returning a null pointer within the *waitForNextDataObject* function. This functionality is achieved via private communications between matching split and merge operations. This section describes the nature of these private communications (Section 2.6.1) as well as additional functionality that they enable (Sections 2.6.2 and 2.6.3).

2.6.1 Notify split and notify merge messages

The execution of merge operations is suspended upon calls to *waitForNextDataObject*. The execution resumes either when a new message is received and delivered to the operation, or when all messages have been received (in which case *waitForNextDataObject* returns a null pointer). This information is provided by a feedback loop between the merge operation and its matching split operation.

More concretely, when a merge operation is instantiated, it locates the matching split operation and sends a special *NotifySplit* message to it. Unmatched split operations that are predecessors of the received message can be identified using additional fields contained in the token carrying the message (see [35], Section 5.3 for more details). The split operation then replies using a *NotifyMerge* message indicating the number of messages it sent during its execution. That message is only sent once the user-provided *execute* method of the split operation completed its execution. This mechanism requires the split operation to be kept alive until the exchange with the merge operation has been performed.

The special notification messages are packaged within tokens like any other message. They are however processed internally by the DPS framework on arrival rather than passed on to the user-provided operation. They also have a higher priority and are placed at the head of the reception queue upon reception.

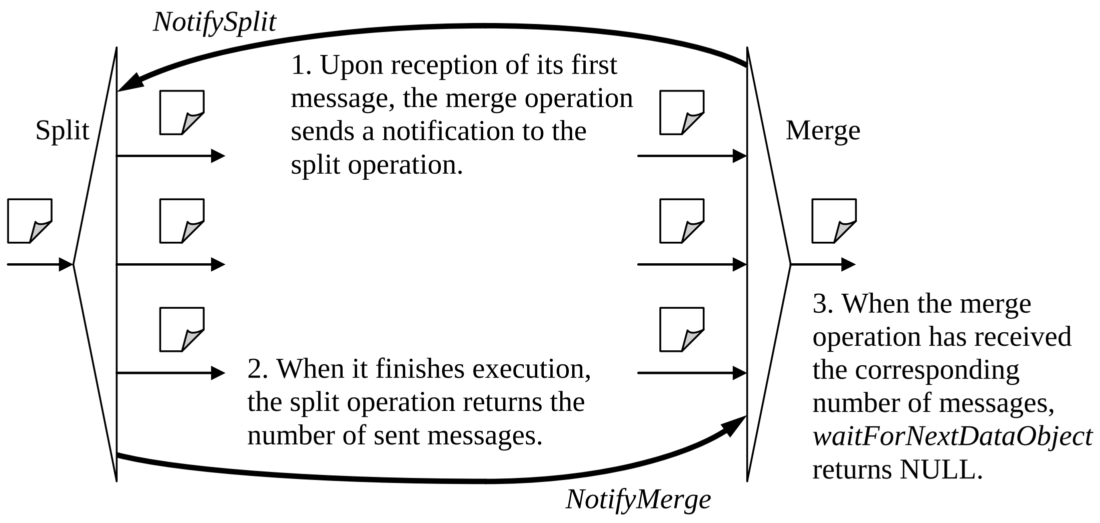

NotifySplit
1. Upon reception of its first message, the merge operation sends a notification to the split operation.
Split
Merge
2. When it finishes execution, the split operation returns the number of sent messages.
3. When the merge operation has received the corresponding number of messages, *waitForNextDataObject* returns NULL.
NotifyMerge

Figure 2.12: *Split-merge communication for determining the merge operation lifetime*

This feedback mechanism also enables DPS to detect routing errors within the application. If the routing function leading to a merge operation is erroneous, messages that should be processed in the same merge might be sent to different threads. In this case, the merge operation will be instantiated once on each thread. The split operation will therefore receive multiple notifications from different merge operations and produce an error.

2.6.2 Flow control

A split operation may create any number of messages, and, due to the asynchronous execution of the flow graph, these messages may accumulate within the target threads and consume large amounts of memory. In some cases, they may also impact the application performance as we will see in Section 3.5.2. In order to resolve this problem, DPS provides a flow control mechanism that limits the number of messages that can be in circulation at a given moment between a given split-merge operation pair.

The flow control mechanism is implemented on top of the *NotifySplit–NotifyMerge* feedback loop described above. When flow control is enabled for a split operation, the split will only send the number of messages indicated in the flow control parameter before suspending its execution. The corresponding merge operation will start the feedback process when it starts receiving messages, indicating how many messages it has already received within the *NotifySplit* notification, thereby freeing the split operation to send more messages.

The default behavior is for a merge operation to send one notification for every incoming message. However, this may not be desirable in cases where the split operation create very high numbers of small data objects. A flow control group size parameter may be used to indicate how many messages should be received before the next notification is sent. This reduces the amount of network traffic created by the flow control mechanism by reducing the frequency with which the merge operation sends notifications to the split operation.

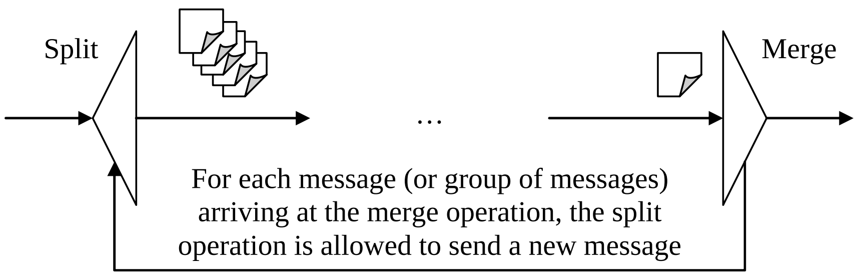

Split
Merge
...
For each message (or group of messages) arriving at the merge operation, the split operation is allowed to send a new message

Figure 2.13: *Flow control in parallel schedules*

The activation and the parameters of flow control are defined individually for each split and stream operation within the flow graph. They may therefore be tuned to the particular needs of each part of the flow graph. If the flow control limit is set to a very low value, some of the threads executing the operations contained within the split-merge pair might have empty message queues during execution. When the queues are empty, the processors are idle, waiting for a new message to arrive. This leads to an underutilization of the available resources. The flow control parameters should therefore be set to ensure that every thread has at least two messages: one being processed within an operation, and a second one waiting for processing.

2.6.3 Load balancing

When using simple compute farm patterns with a routing function such as round-robin routing, load imbalance appears when the distributed tasks are not all of equal complexity or when the individual threads do not have the same processing power available. In order to achieve load balancing, the routing function that distributes the tasks to the threads must send more messages to the threads that can process more operations.

DPS provides such a routing function by taking advantage of the flow control mechanism. Whenever a split operation uses the load balanced routing function, it initially sends out its messages with a round-robin distribution up to the flow control limit. The value for the target thread index is attached to each message, and it is preserved until a successor message reaches the merge operation. The merge operation sends these stored thread indices back to the split operation within the *NotifySplit* message, thereby allowing the split operation to reuse the indices of threads that have already returned results to the merge operation.

This reuse of indices ensures that threads are selected based on their effective computation throughput. The load balancing mechanism can effectively equilibrate uneven computation loads when the number of messages representing subtasks is significantly greater than the number of threads used for computation.

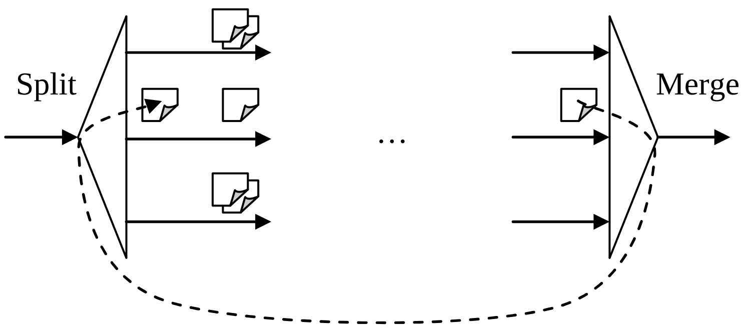


Figure 2.14: *Load balancing in parallel schedules. When the split operation sends out a new message, it is sent to the thread that last returned a message to the corresponding merge operation.*

2.7 Serializable objects

As mentioned in Section 2.4.2, the dynamic instantiation of operations is performed using a class factory. Moreover, the C++ objects used as messages must be serialized in order to transit over the network between operations. Since the C++ language does not provide any serialization or reflection mechanism, DPS provides its own custom mechanisms to enable the same functionality. The mechanism is very easy to use, has high performance, and supports heterogeneous platforms in the sense that an object serialized on one platform can be correctly deserialized on a different platform, taking into account differences in endianness (i.e. in byte ordering) and basic type sizes. It is also able to serialize data in binary and textual form on any medium, such as network sockets, memory buffers and files.

The current section describes how serializable objects are used within DPS, as well as the extensions developed for this thesis. Implementation details can be read in [35], Appendix C. DPS originally used serializable objects in three contexts:

- All messages transfered by the application are serializable C++ objects. An example of a basic serializable object is displayed in Listing 2.1. The key concepts are briefly described in Section 2.7.1.
- Using a serializable object for local thread storage enables checkpointing thread states. This feature is described in Section 2.7.2.
- Operations themselves may be serializable objects, enabling suspended operations to be checkpointed as well (Section 2.7.3).

The original purpose of checkpoints was the development of the fault-tolerance mechanism of DPS, which is described in detail in [35, 36, 38]. In this thesis, checkpointable threads and operations are a key component of the message race detection mechanism described in Chapter 4. Some optimizations described in Chapter 4 for instance rely on the ability to identify the differences between two serializable objects. This extension to the original serialization mechanism is described in Section 2.7.5.

The potential use cases for the automatic serialization of C++ objects go far beyond its current use within the DPS framework and within the current thesis. The functionality was therefore recently packaged as a standalone library called *autoserial*. The library's web site [99] contains exhaustive documentation about declaring and using serializable objects, and provides freely downloadable source code. Appendix B provides a few performance results.

2.7.1 Serializable objects

In DPS, serializable objects must derive from the *dps::AutoSerial* base class. The class must always start with the *CLASSDEF* macro, indicating the name of the type. The next element is the *MEMBERS* macro. The list of members is then given with a set of *ITEM* macros, each taking two arguments: the type of the member variable, and the name. *ITEM*, and the equivalent *PUBLICITEM*, creates public members. These macros simultaneously declare the items: *ITEM(int, a)* is equivalent to *public: int a;*. Private or protected members can be declared using the macros *PRIVATEITEM* or *PROTECTEDITEM*. Finally the declaration of the class is completed with the *CLASSEND* macro. Other macros may be used to create fixed size arrays.

The *ITEM* declarations support all the simple types (int, char, float, etc.), the STL types (std::string, std::vector, etc.), and all serializable types. A *dps::SingleRef* class may be used to serialize pointed objects, and a *dps::Buffer* class provides serializable dynamically allocated memory. The serialization mechanism is able to serialize complex data structures such as trees and circular linked lists.

Serializable classes remain regular C++ classes. The macros only take care of the declaration of serializable members of the class. Methods, constructors and a destructor can therefore be added as needed. Additional member fields may also be declared outside of the macros, with the effect that these members will never be serialized. Hierarchies of serializable classes can be created using the *BASECLASS* macros to identify the base classes of the serializable object.

The described macros enable the serialization to be performed fully automatically. While this is suitable for the majority of cases, the developer may choose instead to write his own serialization and deserialization code if he ever needs to.

2.7.2 Checkpointable threads

As shown in Listing 2.8, the local storage of a thread consists of a single C++ object. Making a thread checkpointable is thus as simple as making the underlying storage class checkpointable. Listing 2.9 illustrates how to make the *VectorThread* class of Section 2.4.5 serializable. Thread checkpoints can then be taken at any time when no operation is running, e.g. between the execution of two operations. A checkpoint also includes all the messages pending in the reception queue of the thread. Recovering a thread state therefore means replacing its current storage object with the one recovered from a checkpoint, and replacing the messages awaiting processing by the ones stored in the checkpoint.

Listing 2.9: *Declaration of serializable local thread storage*

```
// Declare VectorThread thread storage
class VectorThread
{
  CLASSDEF(VectorThread)
  MEMBERS
    ITEM(std::vector<int>, threadVector)
  CLASSEND;
};
```

2.7.3 Checkpointing operations

Even though they do not necessarily perform many computations, split, stream and merge operations may potentially have much longer lifetimes than leaf operations. For instance, a flow controlled split operation must wait for many subsequent operations to complete before being able to send its last message. These long running operations are more vulnerable to hardware failures during the application execution, and DPS therefore incorporates a special mechanism for checkpointing and restarting operations.

Operations may only be checkpointed when they are suspended (Section 2.5.2). The major difficulty lies in maintaining and recovering the stack frame of the operation, such that the user-written code can be resumed. For instance, if a split operation is checkpointed when it calls the *postDataObject* method, recovering the operation from a checkpoint should put it back to a state where *postDataObject* is being called. Unfortunately, stack frames cannot be checkpointed easily, and existing solutions are operating system specific and require external libraries [26, 70]. DPS therefore provides a platform independent mechanism to avoid these issues. The chosen approach is to store the operation state in the operation object and to make the operation class serializable. The stack frame information is not kept in the checkpoint. Operations therefore cannot be recovered as is, and must instead be *restarted*.

When the DPS framework restarts a checkpointed operation, it calls its *execute* method with a null pointer as the input message. It is then the responsibility of the developer to discriminate between the regular execution of the operation and the recovery from a checkpoint. The data stored within the operation checkpoint may be used to restore the operation state properly. For a split operation, such data might include the counter of the loop sending messages, while a merge operation would need to store its partially aggregated data.

Going back to our merge sort example, Listing 2.10 shows how to make the *MergeVector* operation restartable. Compared to the original merge function displayed in Listing 2.4, we

Listing 2.10: *Declaration of a restartable merge operation*

```
class MergeVector
  : public dps::MergeOperation<VectorData,VectorData>
{
  CLASSDEF(MergeVector)
  MEMBERS
    // Serializable reference to output message
    ITEM(dps::SingleRef<VectorData>, out)
  CLASSEND;

public:
  void execute(VectorData *in)
  {
    // Upon first execution, we initialise the output
    // message, otherwise we reuse the checkpointed one
    if (in != NULL)
      out = new VectorData();

    do
    {
      // Upon restart we skip everything up to the call to
      // waitForNextDataObject
      if (in != NULL)
      {
        std::vector<int> tmp;
        tmp.swap(out->v);   // Swap content of tmp and out->v
        std::vector<int>::const_iterator it1 = in->v.begin(),
                                         it2 = tmp.begin();
        while (it1!=in->v.end() || it2!=tmp.end())
        {
          if (it2==tmp.end() ||
              (it1!=in->v.end() && (*it1)<(*it2)))
            out->v.push_back(*it1++);
          else
            out->v.push_back(*it2++);
        }
      }
    }
    while((in=(VectorData*)waitForNextDataObject()) != NULL);
    // Checkpoints may be taken while the merge waits for messages

    out->addRef();
    postDataObject(out);
  }
};
```

only added one serializable member to store the partially merged vector (lines 4–8), as well as two tests to distinguish a regular execution, where the input message pointer is valid, from a restart, where the input message pointer is null. The first test (line 15) ensures that we do not instantiate a new *VectorData* object while we recover from a checkpoint: in that case, the *out* object has already been recovered from the operation checkpoint. The second test (line 22) makes sure that if the operation is being restarted, the operation does nothing but call *waitForNextDataObject* to wait for the next incoming message.

2.7.4 Types of serialization

The serialization mechanism is very versatile. It works by traversing all the members of the object to be (de)serialized. The traversal is performed recursively for cases where objects include or point to other objects. *Writer* and *Reader* objects are then responsible for actually reading or writing the serialized form of each element.

DPS provides readers and writers that can serialize objects as binary streams that may be sent over the network, written to a file or stored in a memory buffer. As a proof of concept, it also supports serialization in XML format, where each member variable is enclosed within <*name*></*name*> tags.

Two additional serializers have been developed for this thesis. The first enables computing the memory size of an object by summing the sizes of all its elements. Compared to serializing objects within memory buffers, it has the advantage of not performing any memory copy and not using additional memory to store the serialized objects. It is used in Chapter 3 in order to estimate the time needed to send a message over the network.

The second serializer writes objects to a TCP socket in a textual form. The serialized stream includes the types and the names of the object members in order to fully describes both the structure and the content of the object. It is used in Chapters 4 and 5 to display the content of serializable objects in a graphical debugger that can be attached to DPS and MPI applications.

2.7.5 Comparing serializable objects

Since we may serialize objects into memory buffers, one may compare two objects by performing a binary comparison of their serialized forms. Nevertheless, we developed a more advanced comparison mechanism based on the member-traversal approach used for serializing objects[4].

[4]See [35] Appendix C.

Listing 2.11: *Comparing serializable objects*

```
class SimpleObject : public dps::AutoSerial
{
  CLASSDEF(SimpleObject)
  MEMBERS
    ITEM(int, intItem)
    ITEM(double, doubleItem)
  CLASSEND;
public:
  SimpleObject(int a, double d) { intItem = a; doubleItem = b; }
};

int main()
{
  SimpleObject s1(1, 0.2);
  SimpleObject s2(1, 0.2001);

  // Declare a comparator
  dps::Comparator c;
  c.printMembersEquality(&s1, &s2);

  // Set higher tolerance for double comparisons
  c.setEpsilonDouble(0.002);
  c.printMembersEquality(&s1, &s2);
  return 0;
}
```

Rather than serializing members, we compare them individually. Compared to a basic binary comparison, such an approach brings several benefits that will be reused in Chapter 4:

– Avoiding the actual serialization avoids storing the compared objects within memory buffers, thereby reducing both the time and memory required by comparisons.

– It provides a much finer resolution since it is able to tell which members are actually different.

– It enables the use of different comparison methods when binary comparisons are too strict. The default mechanism includes support for adding a certain tolerance when comparing floating point numbers (doubles and floats). When declaring serializable classes, the developer may also provide his own custom comparison method.

The notion of equality (or rather of equivalence) can therefore be fully customized by the developer. For instance, we could imagine cases where two lists or arrays should be considered equal when they contain the same elements in different orders. While a binary comparison takes the element order into account, the developer may write his own code to ignore that aspect.

Listing 2.11 illustrates a simple example. We use a *Comparator* object to perform comparisons, and use its *printMembersEquality* method to display the differences between the compared objects. When we execute the program, the function call at line 19 outputs:

```
SimpleObject: modified
  intItem: equal
  doubleItem: modified
```

The members are indented to indicate the level of indirection required to reach them. Setting a sufficiently large ϵ makes the two objects look equal to the comparator.

2.8 Application deployment

Starting a DPS application requires starting one or more processes on each of the participating compute nodes. In the DPS computation model, these processes can be started at any time during the execution of the program, enabling the application to adapt its resource consumption to its actual needs, or to recover processes from node failures.

From the point of view of the developer, all communications are performed between threads. The destination thread of a message is identified by (1) the thread collection to which the next operation is attached, which is determined by the flow graph, and (2) by the thread index, which is computed by the routing function. Within the DPS runtime, a *network layer* fully abstracts all communications. It hides the physical location of the running processes by handling their addressing, and provides high-level functions that take care of the actual data tranfers.

Thanks to these abstractions, the allocation (or *mapping*) of threads onto processing nodes may also be modified during the execution of applications. While this is easy for threads that have no local storage, the availability of checkpointable local thread storage and operations enables threads to be migrated even when complex data structures are stored locally and when the thread still has active (but suspended) operations.

The original implementation of DPS provided a single network layer based on raw TCP sockets, which is described in Section 2.8.2. Two additional network layers where developed in the course of this thesis. The first interfaces with a performance prediction and simulation tool that forms the topic of Chapter 3. The second is a network layer based on the Message Passing Interface (MPI [110]) and is briefly presented in Section 2.8.3.

2.8.1 Thread mapping

The mapping of threads onto processing nodes is performed at runtime, and can be altered using three methods. The first, *addThread* takes a string identifying the application instance (i.e. the process) where threads should be created. Process identifiers are specific to the underlying network layer. The TCP network layer uses identifiers of the form *ipAddress:port* or *hostname:port*, while the MPI network layer identifies processes by their rank. Repeating the same identifier causes multiple threads to be created within the same process, e.g. to make use of multiple processing cores within a single node.

The *removeThread* method enables reducing the size of a thread collection. Ongoing operations may terminate, and messages that are pending in the incoming queue will be processed, but no new message will be routed to a removed thread[5].

Threads may finally be migrated from one process to another thanks to the *moveThread* method. A first parameter specifies the identifier of the thread to be moved, and the second identifies the destination process onto which the thread will be migrated.

All methods are applied to the thread collection enclosing the created, migrated or removed thread. In the example of Section 2.4, Listing 2.5 illustrates the use of the *addThread* method at line 103. In the case where threads are added or moved to a process that does not yet exist, it is the responsibility of the network layer to launch a new application instance to host the thread.

2.8.2 TCP network layer

TCP sockets provide multiple benefits as a network communication layer. TCP/IP is available on most hardware and operating systems and is therefore able to run applications on heterogeneous platforms. For instance, trials have successfully run DPS applications on a "cluster" composed of big- and little-endian CPUs running Linux, Mac OS X and Windows. TCP also guarantees non-lossy FIFO links, i.e. messages are never altered, and they are not reordered during communications between a single sender and receiver pair. Finally, TCP provides full control over how and when network connections are established and data is transmitted. Within DPS, these properties are used for dynamically adding and migrating threads and for running applications on heterogeneous clusters. They also enable a network layer implementation that does not invalidate properties of the parallel schedules approach, namely the asynchronous and deadlock-free execution of flow graphs.

[5]It is therefore the responsibility of the application developer not to remove a thread supporting a partially executed merge operation: such a case would lead to the creation of another operation to process the remaining messages and would be detected as described in Section 2.6.1.

The TCP network layer is able to dynamically start new processes. When adding or moving threads, the network layer checks whether a process is already running or whether it needs to be started. Multiple startup mechanisms exist, that start new processes locally (e.g. for debugging), or remotely. Remote process startup can be carried out using RSH or SSH, or via a kernel running on each of the processing nodes.

2.8.3 MPI network layer

The MPI network layer is a recent addition to DPS. Its current implementation requires that all the processes participating in the computation are launched at once upon startup. This does not prevent applications from adding, migrating or removing threads during their execution, however they can only do so on a predefined set of processes. By using MPI instead of TCP, applications also lose the ability to run on heterogeneous platforms.

On the other hand, MPI offers several benefits, such as faster startup and shutdown times, as well as the ability to take advantage of the native communication protocols of fast networks such as Quadrics, Infiniband or Myrinet. From the perspective of the developer, the only visible change lies in the process identifiers used to add or move DPS threads.

The use of MPI makes DPS applications a special kind of MPI applications. Application must therefore be launched like any regular MPI application, e.g. using the *mpiexec* or *mpirun* launchers. This enables DPS applications to take advantage of multiple tools originally developped for MPI. Of immediate interest to application developers are the use of MPI-aware debuggers. For instance, the *mpiexec* launcher of MPICH2 offers a *-gdb* command line parameter to attach a debugger to all processes upon startup. Integrated development environments plugins such as the Parallel Tools Platform for Eclipse, or the Compute Cluster Pack for Visual Studio offer similar functionality using graphical debuggers.

The availability of MPI also enables DPS applications to use existing MPI libraries such as ScaLAPACK [19, 95] or FFTW [33]. The network layer implementation and other additional functionalities are described in Appendix C.

2.9 Conclusion

The Dynamic Parallel Schedules (DPS) library offers high-level constructs for writing parallel applications. Developers write a set of serial operations, and link them together within a directed acyclic graph, called flow graph, that describes the possible flow of data and the orderings of computations. All message transfers and operation executions are performed asynchronously. The acyclic flow graph construction and the implementation of the DPS runtime guarantee that no deadlock may occur provided that all operations output at least one message.

The actual deployment of the flow graph, including the number of messages sent, is known only at runtime. Loop constructs may be added to the flow graph to dynamically adapt the length and depth of the graph to the data being processed. Operations execute within threads, which are deployed at runtime onto compute nodes. While multiple DPS threads may run within a same process, each thread provides private memory storage to its operations, thereby providing a distributed memory architecture. Threads may be added, removed and migrated during execution.

The vast majority of the DPS code was written by Sebastian Gerlach during his PhD at the Peripheral Systems Laboratory of EPFL [35]. My personal contributions to the core library can be found mostly in the form of extensive testing of the library and of a set of applications on multiple platforms. I also added several extensions, and fixed a few bugs. Other significant contributions are the ability to compare serializable objects and the packaging of the serialization and reflection capabilities as a separate library.

Several internal design decisions have been motivated by the work on parallel application simulations that forms the topic of the next chapter. Much effort was invested into making DPS modular and extensible. The content of the next chapters show that this goal was clearly achieved. Besides the new simulation and testing capabilities described in Chapters 3 and 4, DPS applications are now also able to run on top of MPI (see Appendix C).

Several features presented in this chapter are reused heavily in the remainder of this document. In Chapter 3, simulation and performance prediction capabilities are enabled by the high level of abstraction provided by DPS threads. The versatile serialization mechanism is used to estimate the transfer time of messages over the network.

In order to cause and detect message races within DPS applications, Chapter 4 heavily uses the checkpoint and recovery capabilities of DPS in order to store intermediate application states. The serializable object comparison mechanism is used to determine how states and messages vary in different executions. Chapter 4 also leverages the two great advantages provided by the DPS flow graph. Firstly, its graphical representation is very valuable for representing the application state during debugging. Secondly, it provides some information about future computations which helps identifying equivalent executions. This significantly reduces the cost of detecting message races.

Full details about writing and running DPS applications are found on the DPS website [37].

Chapter 3

Simulation and Performance Prediction

3.1 Introduction

While some problems are inherently hard to solve in parallel, those that are suitable for parallelization are often parallelized in an inefficient way. Devising parallelization strategies that minimize serial computations and communication overheads is key to reaching good performance.

Many parameters have an impact on the execution efficiency. Changing any one of the granularity of the problem decomposition, the mapping of threads to processing nodes, the routing functions or the use of flow control and load balancing may change application running time. The performance obtained with a given set of parameters may also depend on the input data as well as on the number of compute nodes used by the application. To make matters worse, the optimal parameters are often not only application specific, but they also depend on the speed of the network and the processing power of the cluster within which the application is running. As an example, the High Performance LINPACK benchmark [25] used to rate supercomputers on the Top 500 (www.top500.org) provides 16 tunable parameters that can produce millions of different configurations.

Similarly, the load of parallel applications may vary over time, causing some processors to become idle or underused during parts of the execution. In such cases, modifying the allocation of nodes to applications at runtime can help optimize the utilization of resources and compensate inefficiencies within the program parallelization. However, taking good decisions about how and when to modify the allocation of compute nodes requires *a priori* knowledge about the *dynamic efficiency* of the application, i.e. its utilization of computation resources as a function of time.

The number of possible parameter combinations quickly becomes overwhelming, and many test runs must therefore be performed to obtain the necessary information. This testing phase can be time consuming on busy production parallel systems, since jobs must wait until compute nodes become available. Rather than wait for nodes just to run tests, developers generally prefer to run their applications as is, without spending too much effort on their optimization.

There is therefore a need for performance prediction tools that help developers understand the behavior of their applications and identify performance bottlenecks. It is not sufficient to focus on the overall running time of the application. The tool must be able to take into account the task scheduling strategies implemented by the application, reveal runtime information about detailed parts of the execution, and provide information about the dynamic efficiency of the application as well as the effectiveness of the chosen problem decomposition.

It is also important that simulations may be obtained using a single computer in less time that it takes to run the actual application. This enables developers to run their application on their desktop computer or on one cluster node reserved for development and debugging purposes.

3.1.1 The need for optimizations

We studied the utilization of the Pleiades cluster at EPFL. The cluster is composed of 132 single-processor Linux nodes interconnected by a single Fast Ethernet switch, and is mostly used for running plasma physics and fluid mechanics applications. Users submit jobs by booking a fixed number of nodes for a maximum amount of time. A scheduler allocates the nodes to the submitted jobs according to various parameters and using backfilling to reduce idle time. If a job finishes earlier than the allocated time, the nodes are freed for subsequent jobs; otherwise the scheduler kills all running processes in order to let further jobs run. During office hours on weekdays, it also reserves a few nodes for interactive use and for small jobs running for less than one hour. The scheduler logs job information, including the number of nodes booked, the start time and the end time.

We analyzed data collected between April and December 2004 by the *system activity data collector* (*sadc*) of the *sysstat* monitoring tool. The available information about the CPU and network usage of every node has a resolution of 10 minutes, and was correlated with the job

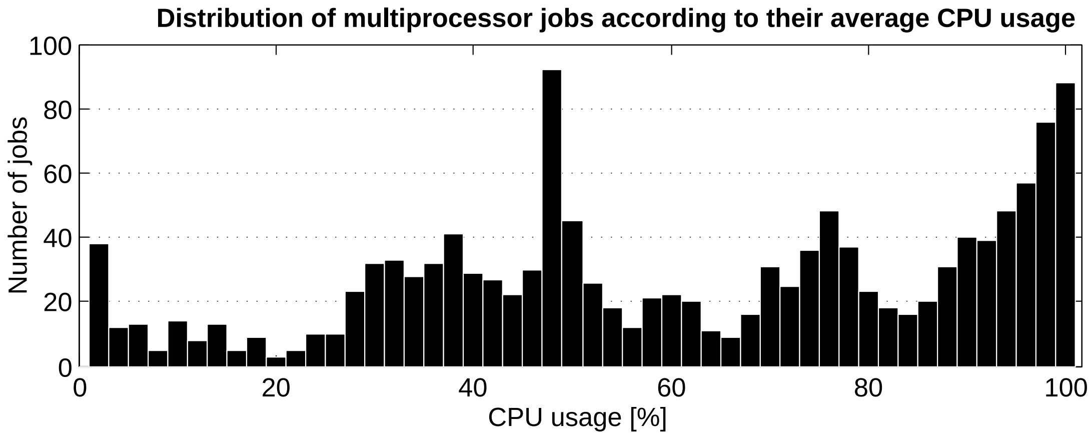


Figure 3.1: *Jobs with all efficiencies have been run on the Pleiades cluster. A significant number use less than half the computing power at their disposal.*

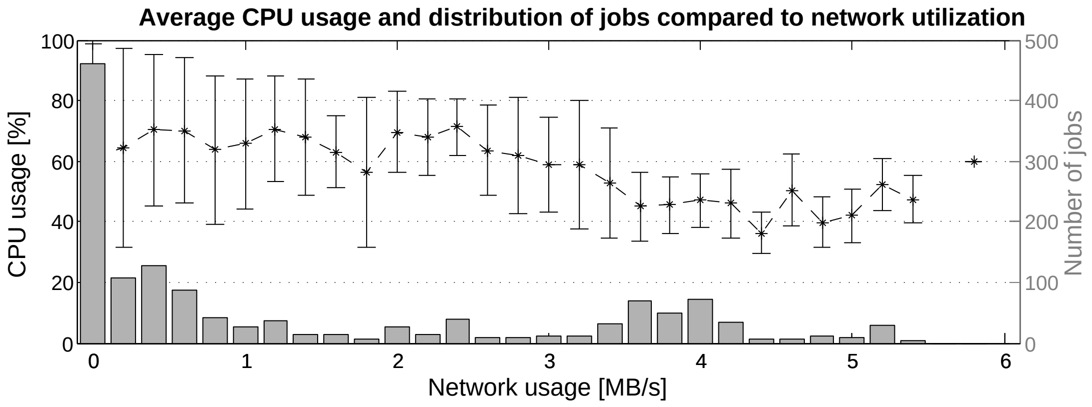


Figure 3.2: *The dotted line displays the average and the variance of CPU utilization of multiprocessor jobs as a function of their average sustained network usage. The histogram displays the number of jobs in each category.*

information provided by the scheduler. Among the 23059 jobs ran on the cluster during that period, we focus here on multiprocessor jobs for which full statistics are available. We also ignore jobs that ran for less than one hour in order to (1) have at least five measurements samples per job and (2) filter out short test runs, interactive jobs and applications that crashed early.

The distribution of the remaining 1368 jobs according to their average CPU utilization is shown in Figure 3.1. The average CPU usage of a job is computed by averaging the CPU utilization samples of all the processors booked. Overall, the average CPU utilization of all jobs was 65%.

Going into slightly more details, we compared the CPU consumption as a function of the average network utilization of each node (Figure 3.2). Unsurprisingly, jobs that communicate more tend to have a lower average CPU utilization. The large variance in CPU consumption displayed by jobs with low network usage may be explained by jobs sending many small messages. As these network transfers are latency bound, they reduces both the network bandwidth and CPU utilization.

The parallel job scheduling systems used on Pleiades allocates a constant number of compute nodes to each job. Applications performing large amounts of serial computations or sharing load unevenly between processors therefore also suffer from reduced CPU utilization even if they do not communicate much. If these characteristics are inherent to the application, adapting the allocation of nodes to the applications' computation needs may thus further increase the utilization of computing resources during program execution.

Since all our measurements are 10 minutes averages, we do not have enough information to definitely determine how large a role is played by each factor. Nevertheless, the limited CPU efficiency of the jobs indicates that easy to use performance analysis and prediction tools would be greatly beneficial. They would help users reduce the running time of their applications, thereby leaving more CPU cycles available for themselves and for other users.

3.1.2 Related work

Much research has already been carried out on predicting the performance of parallel programs with static node allocation. Purely analytical models are generally tailored to a specific application [58] or to a class of parallel programs, such as fork-join applications [85]. Other models have two levels of hierarchy [1], with a higher-level component representing the task-level behavior of the program and a lower-level component representing individual task execution times. These models describe the task-level behavior as a task graph [1, 69] or as a timed Petri net [4]. Approaches for modeling individual task execution times include measurements [4, 58], stochastic models [69, 85] and the association of an application signature and a machine profile [109].

Another proposal for predicting the running time of serial codes compares the memory bandwidth and the number of operations performed by the application with the peak memory bandwidth and the peak processor performance offered by the hardware [57]. This model has been integrated within the IANOS project [89], which provides a performance prediction model for parallel applications that are load-balanced, that do not overlap communications and computations, and whose memory and processor utilization characteristics do not vary over time.

MPI-SIM [86] and its extension COMPASS [7] are two simulators that predict the performance of MPI programs by executing the actual application code. The simulation functionality is provided by a modified library that implements the most common MPI calls. Both MPI-SIM and COMPASS derive computation times through direct execution [22], i.e. by executing and measuring the running time of the application code. The simulation should therefore run on the same hardware as the parallel application. The code does not need to be modified, and no distinct model of the application must be maintained. However, a single processor performs all computations and the whole problem must fit into the memory of a single computing node, thus limiting the size of applications that can be simulated. MPI-SIM and COMPASS alleviate these problems through parallel simulation. A major drawback of this approach is that running the simulation itself requires access to the parallel system. The authors motivate their work by simulating their applications on hardware that is not available, e.g. to guide purchasing decisions.

The aforementioned simulators work for applications that use a fixed set of nodes and processes. The problem of dynamically allocating resources to parallel applications has been considered in other work [21, 54, 117]. Scheduling has also been considered from a theoretical perspective, by modeling parallel applications as Directed Acyclic Graphs (DAG) of serial tasks [16, 63, 127]. According to our knowledge we present the first simulator that predicts the performance of real adaptive applications, i.e. applications whose mapping to computation nodes may vary over time during program execution.

3.1.3 Our approach

We follow a mixed approach, which we call *partial direct execution*. We determine the number and destination of messages as well as the location of operation executions by executing the runtime and application code within the simulator. However, computations that have no impact on the task-level behavior of the application may be replaced by duration estimates. Additionally, we may reduce memory usage by avoiding data structure allocations. The direct execution drawbacks are therefore considerably reduced.

Unlike other simulators which ignore network delays [2, 85], we take network overheads into account by using a simple model and a small set of platform-specific parameters. As a result, our simulator is portable and the execution of parallel programs can be accurately simulated on a desktop computer.

Identifying platform parameters and task duration estimates enables simulations to provide insights about the sensitivity of the application to each parameter. This helps identifying potential performance optimizations as well as determining whether the execution is CPU- or network-bound. Simulations therefore enable application developers to study and improve the performance of their applications without maintaining a separate model and without having access to a parallel machine.

This chapter describes the simulation capabilities that have been integrated into DPS. The integration of the simulator within the framework enables simulating a parallel application by fully or partially executing the application code. This enables reconstructing its exact behavior. Since the simulator also executes the DPS runtime code, features such as the dynamic allocation of processing nodes or the production of an execution trace are also simulated.

Section 3.2 explains the integration of the simulator within DPS. The assumptions made about the parallel system are described in Section 3.3. Section 3.4 details practical considerations regarding the use of the simulator by an application developer. We then show simulator validation results for an LU factorization application in Section 3.5, and for a load-balanced traveling salesman problem solver in Section 3.6. Sections 3.7 and 3.8 respectively show the benefits of the partial direct execution and a detailed sensitivity analysis of the LU factorization application. This sensitivity analysis provides insight about the behavior of the application for different cluster model parameters. Section 3.9 discusses the limitations of the proposed parameterization and execution model, and sketches potential future work.

3.2 Structure of the simulation system

Most of the information needed to reconstruct the execution of a parallel application is only available at runtime. For instance, the execution pattern may be data dependent, and future data distribution decisions may be influenced by intermediate computation results. In addition, parallel programs may implement load-balancing schemes that make it very difficult to predict *a priori* the location of computations and the resulting network transfer patterns. This motivated our decision to integrate the simulation capabilities within the DPS parallelization framework. By directly executing code both from the parallel application and from the framework runtime, the simulator knows the destination of every message, the number of messages sent by each split operation and the current number of processing nodes and threads. The simulator is created upon application startup, and takes control of the application execution by intercepting specific events within the DPS runtime.

The DPS runtime relies on a *Controller* object. The controller is responsible for initializing the network layer, for creating and destroying threads and thread collections, and for launching new parallel schedules. (It is visible to the application developer through the calls to the *getController* method, see Listing 2.6.)

In regular executions, each controller is unique within its application instance. However, controllers are designed such that more than one of them can simultaneously exist within the same process. In order to emulate the deployment of threads onto compute nodes, the simulator uses a modified remote launching mechanism that instantiates a new controller for each application instance that would have been launched in a real execution (Section 2.8.2). The simulator simultaneously maintains a virtual representation of each computing node on which the application is virtually deployed (Figure 3.3). During startup, the controller then instantiates a *simulated network layer* to handle all communications between the virtual nodes. The clean separation into layers within DPS enables all mechanisms that rely on the network layer, such as the transfer of messages or the dynamic allocation of threads, to be used without modifications within simulated applications.

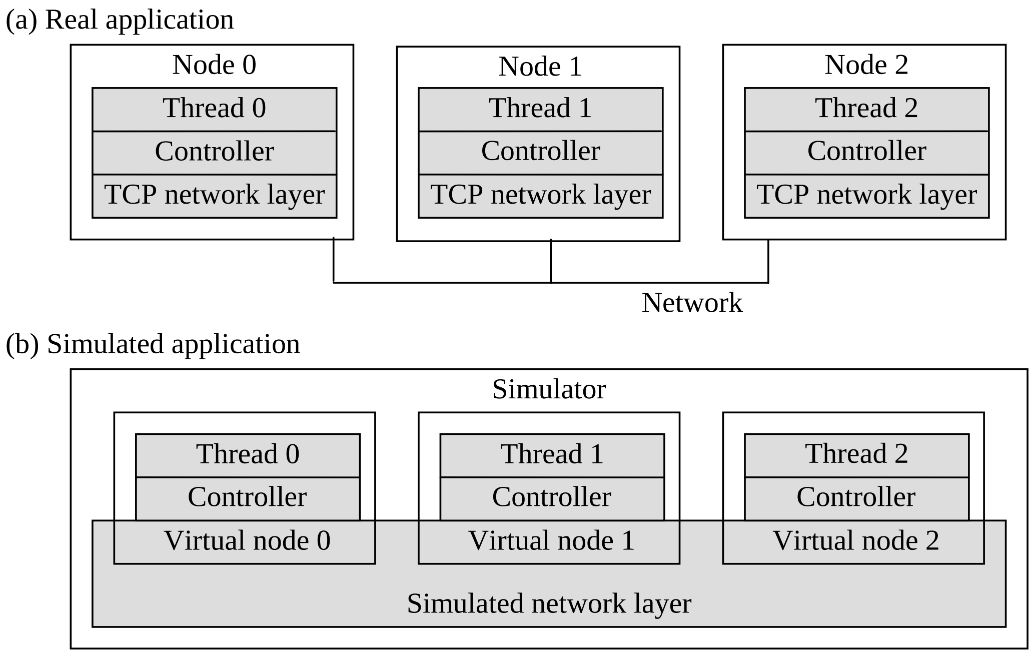


Figure 3.3: *Allocation of threads to computing nodes in a real and simulated application (more than one thread may be running on each node). For the simulation, every thread manager is attached to a virtual node.*

The simulator reconstructs the application execution by keeping track of which threads and which virtual nodes execute the different operations. Since operations may be suspended during their execution, the simulator subdivides them into *atomic steps*, i.e. operation parts which execute without being suspended. Message transfers are also assimilated to atomic steps. Except for the first atomic step of a flow graph, an atomic step starts when another atomic step terminates, and ends when a message transfer completes or when an operation suspends or finishes its execution.

The simulator code runs within its own thread, called the *simulator thread.* In order to avoid any confusion, we will refer to threads that execute DPS operations as *computation threads*. The simulator thread maintains a simulation clock and controls the activation of the computation threads, ensuring that no two computation threads run simultaneously. When a computation thread completes the execution of an atomic step, it queues the atomic step and its duration within the simulator. The computation thread then suspends its execution and resumes the simulator thread. When the simulator thread is running, it advances its simulation clock to the point where an atomic step completes. If the completed atomic step represents a message transfer, the simulator resumes the computation thread that receives the transferred message. If the atomic step belongs to an operation, the simulator resumes the computation thread running that operation. In all cases, the simulator thread and the simulation clock are suspended while the computation thread is running (Figure 3.4).

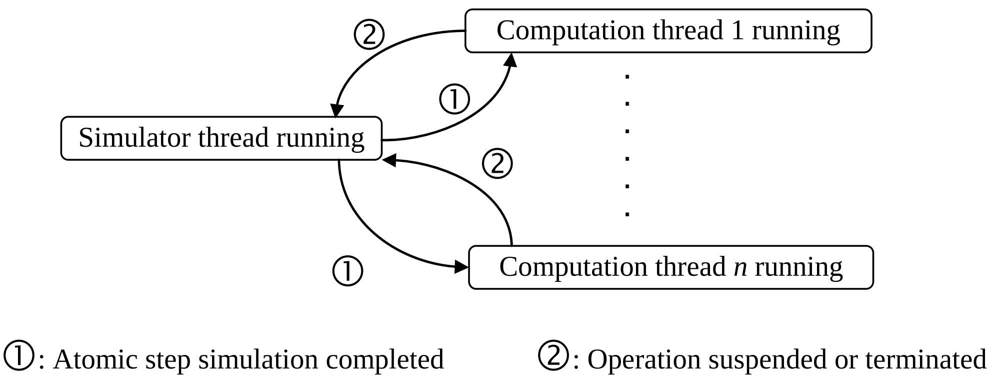

②
Computation thread 1 running
①
Simulator thread running
②
①
Computation thread *n* running
①: Atomic step simulation completed
②: Operation suspended or terminated

Figure 3.4: *State diagram showing the alternating execution of DPS operations and of the simulator.*

Figure 3.5 shows the atomic steps of the execution of a simple flow graph deployed on three threads on three different nodes as in Figure 3.3. One node runs the operations *Split* and *Merge*, while the other two run the leaf operations *Leaf*$_1$ and *Leaf*$_2$. The split operation is composed of the atomic steps S_1 and S_2, which respectively generate the message transfers T_1 and T_2. Each leaf operation consists of a single atomic step, respectively labeled L_1 and L_2. The subsequent message transfers T_1' and T_2' trigger the execution of the atomic steps M_1 and M_2 within the operation *Merge*. The gap between M_1 and M_2 indicates that the *Merge* operation is suspended while waiting for the message from L_2.

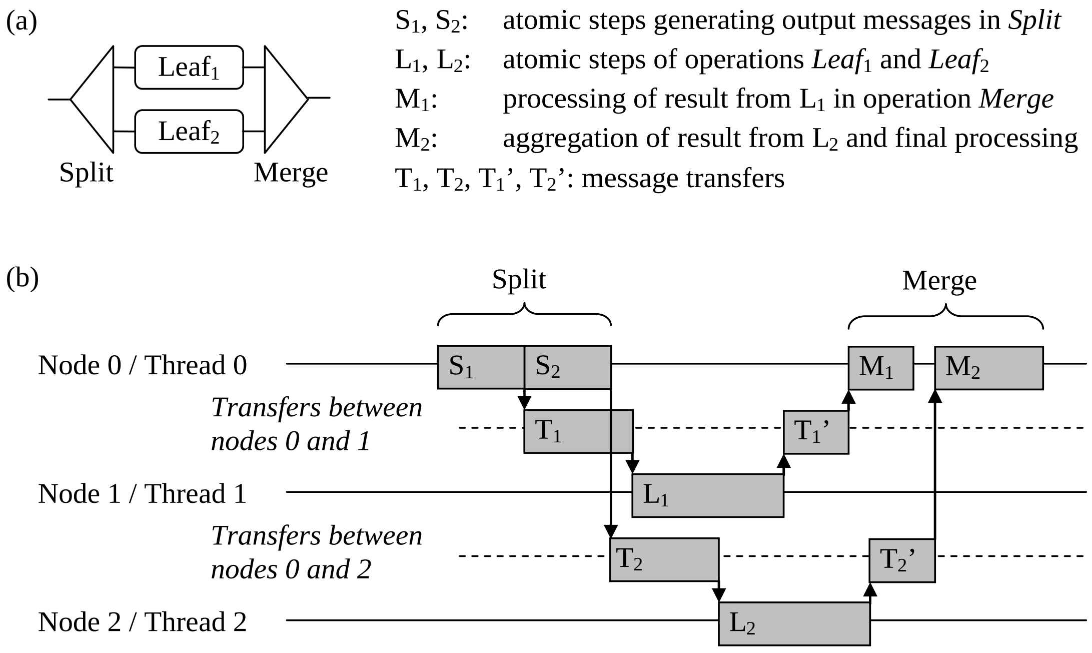

(a)
Leaf$_1$
Leaf$_2$
Split
Merge
S$_1$, S$_2$: atomic steps generating output messages in *Split*
L$_1$, L$_2$: atomic steps of operations *Leaf*$_1$ and *Leaf*$_2$
M$_1$: processing of result from L$_1$ in operation *Merge*
M$_2$: aggregation of result from L$_2$ and final processing
T$_1$, T$_2$, T$_1$', T$_2$': message transfers
(b)
Split
Merge
Node 0 / Thread 0
Transfers between nodes 0 and 1
Node 1 / Thread 1
Transfers between nodes 0 and 2
Node 2 / Thread 2
S$_1$
S$_2$
T$_1$
L$_1$
T$_1$'
M$_1$
M$_2$
T$_2$
L$_2$
T$_2$'

Figure 3.5: *Timing diagram for the parallel execution of a flow graph with a split operation sending two messages to two distinct threads. Each block represents an atomic step. The threads are deployed according to Figure 3.3.*

Figure 3.6 details the temporal execution of the simulation for the flow graph shown in Figure 3.5. The simulator first triggers the execution of the split operation on Thread 0, which runs until the first message is posted. The atomic step S_1 and its duration d_{S1} are queued in the simulator. Control is passed to the simulator thread. Since it did not run yet, the value of the simulation clock is still $t_s = 0$. The simulator thread then increments the simulation clock until $t_s = d_{S1}$, at which point the simulation of S_1 completes and Thread 0 is resumed. Thread 0 first queues the message transfer T_1 in the simulator, and resumes execution of the split operation until the second message is sent and the atomic step S_2 is queued in the simulator. Although T_1 was queued before S_2, both atomic steps start running simultaneously at $t_s = d_{S1}$ in respect to their simulation time. When S_2 completes, control is transferred to Thread 0 which resumes the split operation. Since no other message must be sent, the split operation terminates, and control returns to the simulator thread. The next atomic step to complete in the simulation is T_1. When this happens, the simulator thread delivers the associated message to Thread 1, which is resumed and triggers the leaf operation *Leaf*$_1$. The remainder of the simulation proceeds similarly until the final output message of the flow graph is generated.

The upper part of the timing diagram in Figure 3.6 shows that two computation threads never run simultaneously. The execution of the simulator thread also never overlaps with the

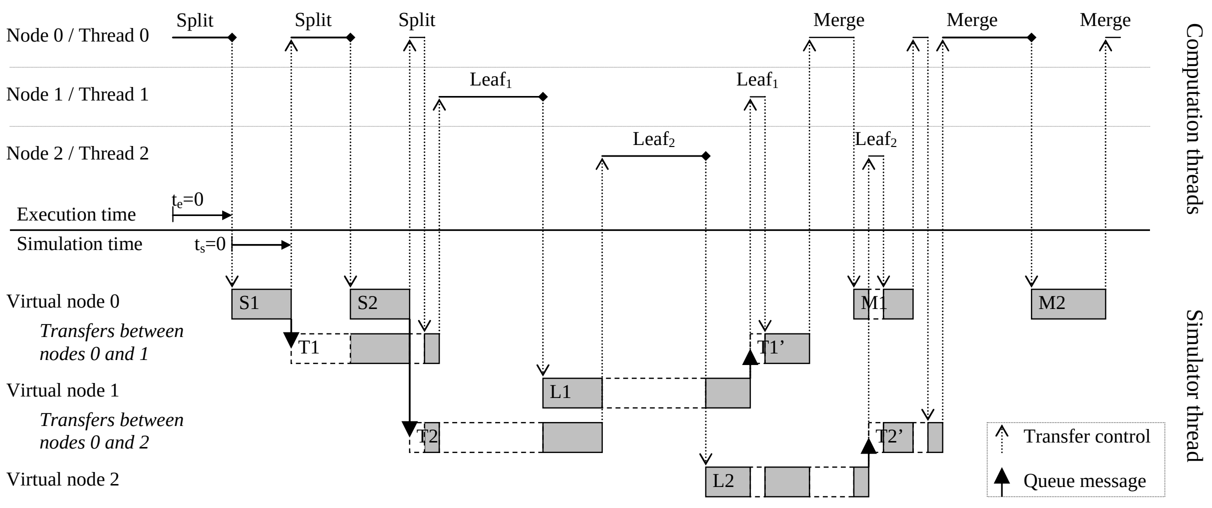


Figure 3.6: *Timing diagram of the simulation of the flow graph shown in Figure 3.5. The upper part displays the execution of the atomic steps that compose DPS operations. The atomic steps are executed one by one, only when the simulator thread is suspended. The lower part shows the management of the simulated time. Removing the dashed gaps between the gray blocks reveals the timing diagram of Figure 3.5.*

execution of the computation threads. In respect to simulation time, operations are correctly overlapping: the timing diagram drawn by the execution of the simulator thread (i.e. with the dashed parts removed) is identical to the timing diagram shown in Figure 3.5. This simulation scheme also requires no a priori knowledge about the execution since the number of messages, their destination thread, and the number and location of operations are all determined at run-time.

3.3 The simulator's system model and its assumptions

In the previous section, we have shown that we can recreate the parallel structure of an application within the simulator given the running time of each one of its atomic steps. Since only a single computation thread is active at any given time, the simulator can measure the actual processing time of each atomic step. The measured time represents the *minimal* duration of the atomic step, which corresponds to its running time when it does not share CPU or network resources with other atomic steps.

For programs whose parallel execution pattern does not depend on the content of the computed data, skipping time consuming computations does not change the destination and number of messages. The prohibitive running time of direct execution simulation may therefore be reduced by using an estimate of the computation time instead of performing the actual computations. We refer to this technique as *partial direct execution*. The time estimate passed to the simulator is simply a number of microseconds that are added to the atomic step running time measured by the simulator to compensate the missing computations.

The source of the estimate is irrelevant to the simulator. It may for instance be deduced from previous executions, computed as a function of some data decomposition parameters, or generated using any other model such as the ones mentioned in Section 3.1.2. By not measuring operation execution times directly, the simulation may also run on a computer that is different and potentially less powerful than the one used for running the actual parallel computations. It is also possible to combine direct execution and partial direct execution. For parallel programs that perform the same operations repeatedly, we may for instance measure the running times of the first n instances of an operation, and reuse the averaged measure for the remaining instances.

The minimal duration of message transfers is estimated using the traditional formula

$$t = l + \frac{s}{b}$$

where l is the network latency, b the peak network bandwidth, and s the size of the transferred message. Although the formula is simple, it is very accurate in predicting the TCP/IP transfer time of messages between two processing nodes and has therefore been widely used [4, 58]. The explicit contribution of l and b also enables determining the individual contribution of each parameter on the overall application performance (Section 3.8). This model however assumes that no network contention occurs, and can therefore underestimate communication costs for network intensive applications. The size of each message is determined by the simulator at runtime thanks to the specialized serializer that computes the size of serializable objects (Section 2.7.4). The latency and bandwidth parameters are constant for a given parallel machine, and must be measured or estimated separately for each target cluster.

We model resource sharing as follows. We assume that the communication network between the nodes has a star topology, where each node is connected via a full duplex link to a central full crossbar switch which is never a bottleneck. The input and output bandwidth are both identical and equal to b. The bandwidth of each node is shared equally among all incoming, respectively outgoing data transfers. A similar model (with arbitrary topologies) was used in [47]. Transfers between operations running on the same thread or between threads running on the same node are considered to be instantaneous.

Since computations and communications may overlap, the processing power used to handle communications also needs to be taken into account. Receiving messages induces more hardware interrupts and more memory copies than sending messages, and is thus more costly. Moreover, we noticed that the consumed processing power depends on the number of outgoing and incoming communications. Similarly to the bandwidth and latency parameters, the processing power required for communications must be measured separately and provided to the simulator. In all cases, the characterization of these communication and processing parameters is independent of the simulated applications, and thus needs to be carried out only once.

We assume that all nodes have a single processor and that no swapping occurs between memory and disk. Since the simulator has a complete knowledge about ongoing computations and communications, it knows at every time point how many concurrent transfers are carried out by each processing node, and therefore the amount of processing power still available. That processing power is shared evenly among concurrently running operations. The simulator also produces detailed statistics about the CPU and network usage of each node during application execution.

In summary, our cluster model takes the network latency, the network bandwidth and the CPU usage of communication as fixed hardware parameters, and assumes that both the CPU of a single node and the bandwidth of a single link are shared equally among ongoing atomic steps.

3.3.1 Internal implementation

The most common technique for performing simulations is *discrete-event simulation*. The execution of the simulated system is represented as a chronological sequence of events. Every event represents a change in the state of the system, and has an associated timestamp. The simulator clock advances by jumping from one timestep to the next. In our context, an event represents the end of an atomic step. (Since the beginning of an atomic step coincides with the end of a previous atomic step, it does not need to be represented separately.) When the simulator processes an event, it computes the completion time of the triggered atomic steps and adds the associated events to the event queue. In case the new atomic steps share resources with atomic steps whose completion time is already queued, the simulator must also update the latter.

A variant, which we refer to as *timeslice-based simulation* is to represent the system execution as a sequence of tasks. The simulator keeps track of the ongoing tasks (or in our case, atomic steps) and iteratively increments a clock with a predetermined time quantum. For every timestep, the simulator removes the elapsed time from the remaining duration of every atomic step. When the remaining duration of a task becomes nil, the task completes and the associated event is processed by the simulator. If several tasks are competing for the same resource, one approach is to subtract a full quantum from one task at every timestep, and to serve the competing tasks in a round-robin manner over multiple timesteps. The other approach is to share every quantum among competing tasks.

Discrete-event simulations have the advantage of having arbitrary time resolution since time hops from one event timestamp to the next. In contrast, the choice of the time quantum is important for timeslice-based simulations. Choosing a timeslice too large reduces the precision of the simulation; choosing a timeslice too small greatly increases the number of iterations that must be performed. Also, the simulation time not only depends on the number of events, but also on the duration of the actual execution.

On the other hand, it is easier to implement resource sharing with a timeslice-based simulator. This last reason is particularly important in our case since all resources are shared. Sending one additional message may potentially cause all queued event timestamps to be recomputed. In Figure 3.7 for instance, sending a message from A to B increases the duration of the transfers from A to C and from C to B. Since network communications consume CPU power, the completion time of all ongoing computations must also be recomputed.

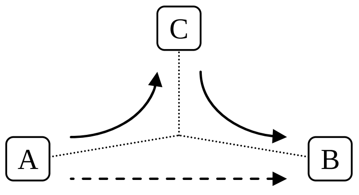


Figure 3.7: *Message transfers are in progress from A to C and from C to B, both using the full network bandwidth. Sending a message from A to B reduces the bandwidth available to other communications, slowing down both transfers and requiring recomputing their completion time.*

This latter reason motivated our decision to implement a timeslice-based simulator. All results in the following sections are produced with a fixed time quantum of 100 microseconds, which is shared among competing atomic steps at every timestep.

One may however use adaptive time quantums in order to reduce the number of timesteps and/or to increase the resolution of the timer. Since we share the time quantum of a given timestep among consuming atomic steps, the quantum size may change from one iteration to the next without introducing a bias in the allocation. We may therefore allocate a quantum as large as the shortest remaining duration among all atomic steps, with the guarantee that we will not subtract more time than is remaining in any atomic step. If the atomic step does not share resources with others, it completes in a single step. Otherwise, a fraction of the quantum is subtracted. In the latter case, the remaining duration of the shortest atomic step decreases at every timestep, and so does the quantum size. Using a minimum quantum duration ensures that the quantum never becomes nil and that the next event is eventually processed. The results presented in this section do not take advantage of this optimization however.

3.3.2 Computing the network bandwidth allocation

For general network topologies and routing schemes, computing the amount of bandwidth available to every message transfer is equivalent to solving a multi-commodity flow problem [28]. In our case however, we only consider network topologies where all nodes are connected via a single full crossbar switch. We may thus devise a simple algorithm to compute at every time step the share of the timeslice (which is equivalent to the share of the bandwidth) available to each atomic step, i.e. to each ongoing message transfer (Algorithm 1). The algorithm assumes that senders and receivers are able to adapt their transmission rate to changes without delay.

As described above, we assume that the available incoming and outgoing bandwidths are

Algorithm 1: Timeslice allocation to communication atomic steps

```
Let $\Delta_t$ be the time quantum and $N$ be the set of computation nodes
Let $L_i^I$ and $L_i^O$ be the set of input, ongoing resp. output message transfers occupying the
incoming, resp. outgoing link of node $i \in N$
Let $L = \bigcup_{i \in N} \{L_i^I, L_i^O\}$, with the sets in $L$ sorted by decreasing cardinality
Mark all atomic steps as unprocessed

foreach $L_j^* \in L$ do
    Let $U_j^*$ be the subset of $L_j^*$ containing the atomic steps marked as unprocessed
    Let $\Delta_{st}$ be the time already allocated to the atomic steps marked as processed (we
    therefore have that $\Delta_{st} < \Delta_t$)
    $\delta = \frac{\Delta_t - \Delta_{st}}{|U_j^*|}$          // Time available to each unprocessed transfer
    foreach $u \in U_j^*$ do
        Subtract $\delta$ from the remaining duration of $u$
        Mark $u$ as processed
    end
end
```

shared equally among ongoing transfers, i.e. atomic steps. However, the bandwidth available for a single transfer may be different between the outgoing link of the source node and the incoming link of the destination node. Algorithm 1 therefore sorts all links according to the number of ongoing transfers, and processes links from the most busy to the least busy.

Figure 3.8 illustrates the behavior of the algorithm. In (a), it first processes the busiest link, i.e. the incoming link of node C L_C^I. Since none of the three transfers of L_C^I have been processed, the bandwidth is split evenly among them. Their remaining duration is therefore decreased by $\Delta_t/3$ at every timestep. Since all transfers have been processed, the algorithm stops and the bandwidth still available on the outgoing link of nodes A, B and C remains unallocated. An additional transfer from B to D (Figure 3.8b) causes the algorithm to recompute the bandwidth allocation. As the busiest link remains L_C^I, existing transfers see no change in their bandwidth allocation. The next busiest link is the outgoing link of B, L_B^O, with two ongoing transfers. The transfer from B to C has already been processed when allocating the input bandwidth of C, so the cardinality of U_B^O is one. The time Δ_{ts} already allocated to the processed transfer is $\Delta_t/3$. The algorithm therefore subtracts $\delta = (\Delta_t - \Delta_{st})/|U_B^O| = 2\Delta_t/3$ from the remaining duration of the unprocessed transfer, reflecting the fact that it has two thirds of the total bandwidth at its sole disposal.

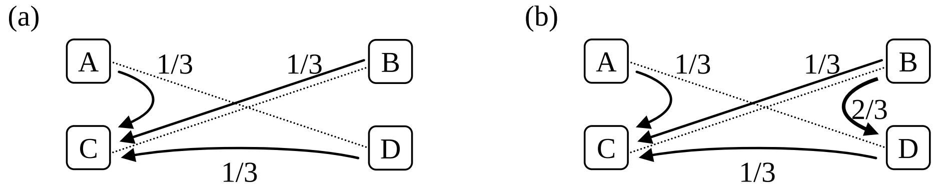


Figure 3.8: *(a) Each transfer only has a third of the available bandwidth. (b) An additional transfer starts from B to D, causing the algorithm to recompute the bandwidth available to each transfer. It processes the busiest link (i.e. the incoming link of node C) first, and splits the bandwidth in three. The second busiest link, the outgoing link of B, has a single unprocessed transfer that may use the whole bandwidth that has not yet been allocated, i.e. two thirds of the original bandwidth.*

The fact that we allocate bandwidth to busy links first ensures that the bandwidth already allocated to processed atomic steps on a given link is always smaller than the total amount of bandwidth available. In Algorithm 1, this translates into the fact that Δ_t is always larger than Δ_{st}. This scheme therefore ensures that (1) we never allocate more bandwidth than is available and (2) the atomic step associated to each transfer gets a share of the bandwidth at every timestep.

3.4 Practical considerations

The code handling the simulation is activated at compile time. Preparing a DPS application for simulation therefore requires recompiling both the DPS framework and the application. Recompiled applications may then be executed as usual, with the same command line parameters.

Network and hardware parameters are specified in a configuration file which is read at the beginning of the simulation. The required parameters are the latency of the network (in microseconds), the peak network bandwidth (in MB/s) and the percentage of CPU consumed while sending and receiving data. If the maximum number of simultaneous communications is n, the amount of CPU consumed is specified as $(n+1)^2$ values indicating all possible combinations from 0 senders and 0 receivers up to n senders and n receivers.

The configuration file can be written by hand with arbitrary parameters in order to simulate fictitious hardware, or be filled with measurement results. The DPS distribution includes a tool that automatically measures these values and produces such configuration files. It is described in Appendix A.

Once a cluster parameterization file is available, running the recompiled parallel application simulates its parallel execution. The recompiled application then runs locally under the control of the simulator, and the predicted running time of the simulated application is outputted at the end of the execution. DPS includes a *Timer::get* method to perform timing measurements. When running within the simulator, this function returns simulation times, such that time measurements performed within the application (e.g. to get the running time of each iteration of a computation) can be used as is. Thread mappings (Section 2.8.1) are specified as for the TCP network layer (Section 2.8.2). Since the simulator assumes that each compute node has a single CPU, all threads mapped on the same host share the same processing and networking resources.

3.4.1 Direct execution simulation

The simulator executes the actual application code and measures the running time of each operation (Section 3.3). Applications can therefore be simulated without a single modification to their source code, with the following limitations:

- The *host* computer, i.e. the computer running the simulation, must contain hardware equivalent to the one available in the *target* parallel machine, i.e. the cluster being simulated.

- The memory consumed by the simulation is the sum of the memory consumed by every application instance on every processing node during a real execution.
- The running time of the simulation can be as large as the parallel running time multiplied by the number of processing nodes used by the application.

These issues limit the size of the applications that can be simulated and tie simulations to very specific hardware. However, all of them may be mitigated using *partial direct execution*.

3.4.2 Partial direct execution simulation

In many applications, the parallelization pattern (i.e. the number and localization of executed operations, as well the size of the transferred messages) is independent of the actual computation results. For instance, the number of block multiplications performed in a full matrix multiplication application only depends on the number of blocks, and not on the content of the matrix. When this is the case, computations do not need to be performed, and can be replaced by a notification of their running time to the simulator. Doing so not only reduces the time and memory required by the simulation, but it also makes the simulation portable. Indeed, running times may now be specified independently of the hardware on which the simulation is running.

Let us take an operation that multiplies a square matrix stored in its local thread storage with another square matrix received within a message. The result of the multiplication is sent off to the next operation (Listing 3.1).

Listing 3.1: *Simple matrix multiplication function*

```
void execute(MatrixData *in)
{
  MatrixData *out = new MatrixData();
  // Output matrix is allocated by the multiplication function
  out->matrix = matmul(getThread()->matrix, in->matrix);
  postDataObject(out);
}
```

Now let us assume that for matrices of size $n \times n$ we can approximate the running time of the *matmul* function on the target hardware using the polynomial $t(n) = 2n^3 + 3.1n^2$. We may use this model to inform the simulator of the running time of the skipped computation using the *addComputationTime* method, which takes a number of microseconds as sole parameter. The operation is modified as follows:

Listing 3.2: *Matrix multiplication operation instrumented for partial direct execution*

```
1  void execute(MatrixData *in)
2  {
3    MatrixData *out = new MatrixData();

5  #ifndef DPS_SIM
6    out->matrix = matmul(getThread()->matrix, in->matrix);
7  #else
8    // Pass computation time to simulator
9    int n = in->matrix->getSize();
10   addComputationTime(2*n*n*n + 3.1*n*n);
11   // We must now allocate a matrix of size n x n, as matmul
12   // would have done
13   out->matrix.resize(n);
14 #endif

16   postDataObject(out);
17 }
```

The final matrix allocation at line 13 is particularly important as the DPS runtime must be able to compute the size of the serialized message in order to predict its tranfer time. Moreover, the operation receiving the *out* message most probably assumes that the matrix is allocated.

Calling *addComputationTime* does not stop the simulator from measuring the operation running times; its parameter is simply added to the measurement performed by the simulator. While this implies that the hardware is only partially abstracted, simply ignoring the time spent in the rest of the operation would reduce even more the accuracy of the estimation. The time required to obtain a running time prediction is therefore also taken into account when the simulator determines the duration of an atomic step. Since it is quite rare that we are able to approximate a computation duration using a simple function (discontinuities may for instance appear when the processed data no longer fits in the CPU cache), it may be necessary to use more complicated models, a hardware simulator or to read benchmarks from a file. The time required to obtain such predictions may thus artificially inflate the duration of an atomic step. In such cases, we may also use the *addComputationTime* function to subtract delays induced while obtaining the prediction.

Listing 3.3: *Subtracting artificial overheads*

```
  long long start = dps::Timer::getReal(); // Returns the wallclock time
  long long computationTime = getPredictionFromModel(in->matrix);
  long long overhead = dps::Timer::getReal() - start;

  // Subtract time required to obtain prediction
  addComputationTime(computationTime - overhead);
```

There is obviously no strict rule specifying which parts of an operation should be executed and which parts should be skipped. For instance, if a set of functions is used in multiple operations, it may be worth modeling each function separately and replacing each function by a call to *addComputationTime*. In other cases it is more convenient to model the running time of a whole operation at once. When computations cannot be modeled easily, it remains possible to use benchmarks of the computations. DPS provides a few helper classes and macros to automate the collection and retrieval of such measurements (Appendix A).

When the parallelization pattern depends on the computation results, it may not be possible to avoid performing the computations. In such cases, the *addComputationTime* method may still be used to artificially lengthen or shorten an atomic step duration to simulate an execution on different hardware. This approach makes the simulation portable, but still suffers from the two other drawbacks of direct execution, namely, the excessive running time and memory consumption of the simulation.

3.4.3 Avoiding memory allocations

When the simulation runs without performing computations, it may become possible to avoid the allocation of certain parts of thread states and of data objects. Going back to our matrix multiplication example in Listing 3.2, skipping the actual multiplication computation implies that the content of the matrices is neither read nor written. The matrices stored in the thread, as well as the ones stored in the input and output messages do thus not need any memory allocation.

Unlike in Java for instance, where arrays have an associated *length* field, the size of C++ memory buffers must be stored separately from the buffer. When computing the size of the matrix objects, the serializer reads the size descriptor without accessing the memory buffer. It is therefore possible to fake the size of a buffer and of the enclosing object by setting the size descriptor without actually allocating the memory, and without altering the network transfer time predictions of the simulator. By avoiding allocating large data structures, one greatly reduces the memory requirements of the simulation (the running time of time consuming memory operations can be explicitly added if necessary).

In DPS, the only built-in class able to serialize dynamically allocated memory provides a *resize* method for the allocation. During simulations, the developer may instead use *simResize* to update the internal variable storing the buffer size without performing any allocation. User-defined objects using custom serialization methods may provide similar hooks.

3.5 First test application: LU factorization

We first measure the accuracy of our simulator for a parallel block LU matrix factorization application with partial pivoting [40]. The block-based LU factorization relies on the iterative decomposition of the matrix. Pipelined implementations loosen the synchronization between the successive iterations and improve the interleaving of operations belonging to successive iteration steps. Such modifications only influence the ordering of the computations, and have no impact on the total amount of data transferred over the network, on the location of the operations, or on the amount of computation they perform. The amount of parallelism and the decomposition granularity of the problem can also be varied, so as to produce executions with different communication patterns and with different computation to communication ratios. Since the amount of computations decreases with every iteration, the efficiency of the application varies over time and can benefit from a reduction in the number of allocated compute nodes. The application therefore provides a wide range of runtime behaviors.

Efficient implementations of the parallel LU factorization use a block-cyclic distribution [19] rather than the parallelization strategy described below. Nevertheless, the higher network utilization of our implementation makes it a good candidate for validating our resource sharing assumptions.

3.5.1 Implementation

Consider a matrix A of size $n \times n$ and a block size r such that r divides n. We want to decompose the matrix A into a product of a lower-triangular matrix L and an upper-triangular matrix U. In order to guarantee the numerical stability of the result, we actually compute $A = P \cdot L \cdot U$ where P is a permutation matrix, i.e. a row permutation of the identity matrix.

The matrix A is split as follows:

$$A = \begin{array}{cc} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & B \end{pmatrix} & \begin{array}{c} r \\ n-r \end{array} \\ \begin{array}{cc} r & n-r \end{array} & \end{array} \tag{3.1}$$

This matrix is decomposed as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & B \end{pmatrix} = P \begin{pmatrix} L_{11} & 0 \\ L_{21} & X \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & Y \end{pmatrix} \tag{3.2}$$

According to this decomposition, the LU factorization can be realized in three steps.

Step 1. Compute the rectangular LU factorization with partial pivoting.

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} \cdot U_{11} \tag{3.3}$$

where L_{11} and U_{11} are lower and upper triangular matrices respectively.

Step 2. Compute U_{12} by solving the triangular system

$$A_{12} = L_{11} \cdot U_{12} \tag{3.4}$$

This is the operation performed by the **trsm* routine in BLAS [65]. Carry out row flipping according to the permutation matrix used of step 1.

Step 3. At this step, L_{11}, L_{21}, U_{11} and U_{12} are known. Completing the LU factorization requires that X be lower triangular and Y be upper triangular. We therefore define $A' = X \cdot Y$, and iteratively apply the block LU factorization to A' until A' is a square matrix of size $r \times r$.

$$\begin{aligned} B &= L_{21} \cdot U_{12} + X \cdot Y \\ A' &= X \cdot Y = B - L_{21} \cdot U_{12} \end{aligned} \tag{3.5}$$

Our implementation uses two thread collections. The first contains n/r threads, each of which stores one column block of size $r \times n$. The second thread collection is dedicated to performing the multiplications of matrix blocks. It may contain any number of stateless threads. The flow graph for the LU decomposition is shown in Figure 3.9. Operation (a) performs the LU factorization of the top left block A_{11} (Step 1), and (b) solves in parallel the triangular system in order to compute U_{12} for all other column blocks and performs the row flipping (*trsm*, Step 2). For the LU factorization presented here, the most expensive part both communication- and computation-wise is the block-based matrix multiplication $L_{21} \cdot U_{12}$. The blocks from L_{21} are available on the local thread within which the stream operation (c) is executing, and the $r \times r$ blocks from U_{12} are transferred from the local thread states where the preceding *trsm* operations (b) were carried out. The messages sent to each of the matrix block multiplications (d) contain two matrix blocks of size $r \times r$. Messages are routed such that multiplications are evenly distributed on all threads. Each matrix block multiplication yields an $r \times r$ matrix block that is sent to the subtraction operation (e) (Step 3). Notifications are collected at the end of

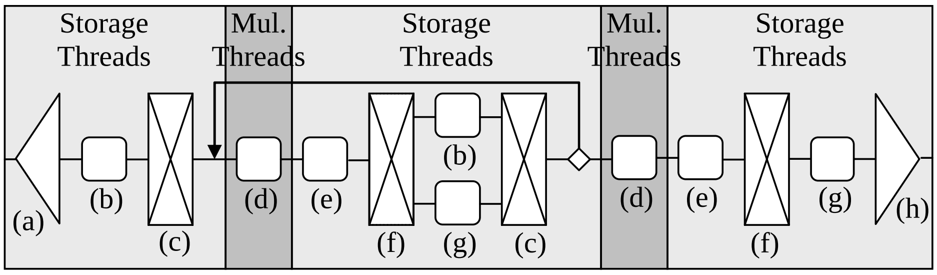


Figure 3.9: *Flow graph for the LU factorization. (a) LU factorization of top left block A_{11} (step 1); (b) parallel triangular system solve for computing U_{12} and perform row flipping (step 2); (c) collect notifications of finished triangular system solve and stream out multiplication requests; (d) block-based matrix multiplication for $L_{21} \cdot U_{12}$; (e) subtract multiplication result from B (step 3); (f) perform next level LU factorization; (g) perform row flipping on previous column blocks.*

the multiplications, and as soon as the first block is complete, the next level LU factorization is performed (f). Triangular system solve requests are streamed out to other column blocks as other column blocks complete, while operation (g) performs the row flipping on previous column blocks. The recursion on the matrix factorization is obtained by replicating a part of the graph (in gray) once for each LU factorization level. Once the last block has been factorized the merge operation (h) collects row exchange notifications for termination.

Figure 3.10 displays the unfolded flow graph when the matrix is decomposed into four column blocks. Due to the construction of the flow graph, two iterations of the flow graph loop are sufficient to produce the four iterations required to fully decompose the matrix. Since the *trsm* operations (dark gray, operation (b) in Figure 3.9) and the row flipping operations (light gray, operation (g) in Figure 3.9) are executed on subsequent and previous column blocks respectively, there is one fewer *trsm* and one more row flipping operation at every iteration.

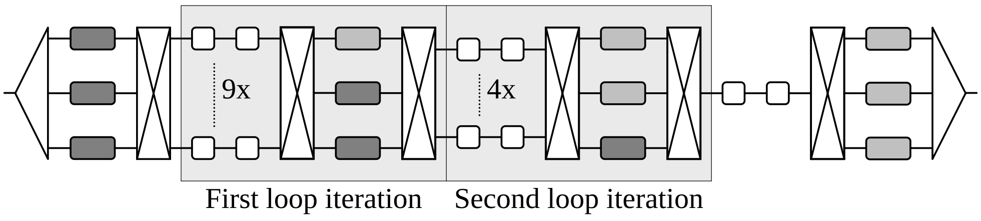


Figure 3.10: *Unfolded LU factorization flow graph with a matrix decomposition on four column blocks, i.e. with four storage threads. Dark gray operations are* trsm *operations, and light gray operations are row flipping operations.*

3.5.2 Variants

We now explore variations of the decomposition block size, modifications of the LU factorization flow graph and the use of the flow control mechanism provided by the DPS parallelization framework.

P variant We refer to the flow graph of Figure 3.9 as the *pipelined flow graph*. Indeed, the stream operations (c) and (f) increase the pipelining of the application, i.e. the number of operations that may run concurrently, by allowing *trsm* and *LU* operations (b) and (f) to be performed simultaneously with matrix multiplications (d) and their associated data transfers.

Basic variant We produce a less efficient *basic flow graph* by preventing pipelining. This is achieved by replacing stream operations with merge-split pairs of operations, thereby introducing barrier synchronizations.

Varying the block size r used for the decomposition has an impact on the number of operations, and consequently on the computation to communication ratio of the application [24]: smaller blocks yield a lower computation to communication ratio. In the pipelined flow graph, the value of r also influences the depth of the pipeline, and thus the amount of overlapping that can be achieved.

FC variant Each thread has an associated queue that stores incoming messages until they are processed (Section 2.5). Sending all multiplication requests at once thus fills the queues of the destination threads, which delays the processing of requests sent by subsequent iterations and reduces the pipelining potential. By applying flow control to the stream operations that generate the multiplication requests, we limit the number of messages queued at each iteration. This improves the pipelining by interleaving operations belonging to successive iterations (Figure 3.11).

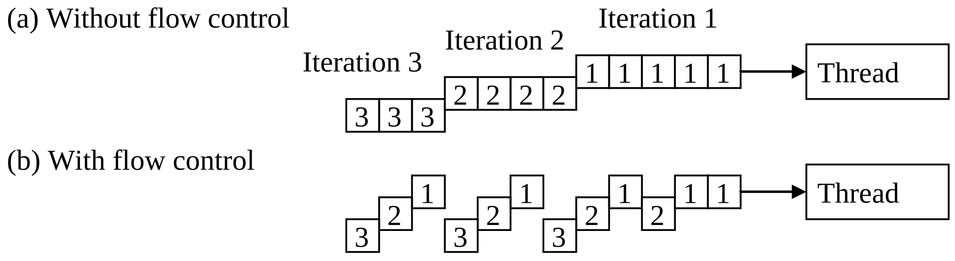


Figure 3.11: *Activating the flow control mechanism improves the interleaving of messages and enables iterations 2 and 3 to be started earlier.*

PM variant Another modification on the LU factorization flow graph consists in further parallelizing matrix block multiplications by decomposing blocks of size $r \times r$ into row blocks of size $s \times r$ and column blocks of size $r \times s$. We use a flow graph (Figure 3.12) that (a) distributes the column blocks of the second matrix to the processing nodes, which (b) store them locally. Each sub-block multiplication can then be performed by (d) sending the line blocks of the first matrix to the processing nodes, which (e) multiply them with the locally stored column blocks. The compositional nature of DPS allows us to replace operation (e) in Figure 3.9 by the flow graph shown in Figure 3.12.

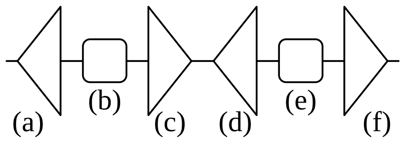


Figure 3.12: *Flow graph for matrix multiplication. It may replace operation (e) in Figure 3.9.*

3.5.3 Validation

We validate the simulator by comparing measurements and simulations using the parallelization and pipelining flow graph variations discussed in the previous section. By combining one or several of the modifications proposed and observing their impact on the parallel application running time, we verify how precisely the proposed network and processing models take the different execution parameters into account.

All the measurements shown below consider the LU factorization of a 2592 $\times$ 2592 matrix carried out either on four or on eight processing nodes. The machines are Sun workstations with a single 440 MHz UltraSparc II processor connected to a full crossbar switch through a Fast Ethernet network. The hardware parameters of the two clusters used for validating the simulator were produced by the cluster parameterization tool described in Appendix A.

In order to compare the different parallelization strategies, we use the *relative performance improvement* metric, defined as the execution time of the basic flow graph (reference time) divided by the execution time of the program incorporating one or several of the proposed variations.

In Figure 3.13, we show the effects of the various deployment and parallelization variants. The reference time (259.4s) is obtained by splitting the matrix into four blocks of 648 columns, distributed on the four available nodes, and by running the application with the basic, non-pipelined flow graph. The parallel sub-block multiplications (*PM*), pipelining (*P*) and flow

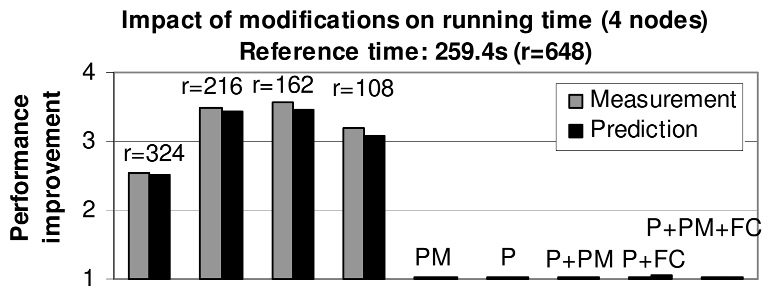


Figure 3.13: *Measured and simulated variation of computation time for the proposed modifications (4 nodes). The reference time is measured using the basic flow graph (no pipelining) when the matrix is split into one column block per node.*

control (*FC*) optimizations bring little improvement (around 3%) to the application running time. Figure 3.13 clearly shows that these gains are negligible compared to those achieved by simply changing the decomposition granularity. Splitting the matrix into sixteen column blocks (r = 162) distributed evenly among the four compute nodes yields the shortest measured and predicted running time, respectively 72.5s and 75.5s. The improvement predicted by the simulator is within a few percents of the measured improvements.

Figure 3.14 shows the effects of the parallel sub-block multiplications (*PM*), pipelining (*P*) and flow control (*FC*) modifications when the matrix is split into eight block columns (i.e. two per node) instead of four. The reference time is the measured running time when r = 324 in Figure 3.13. Due to the well balanced distribution of block multiplications within the reference setup, the increased communication requirements of transmitting sub-blocks for the parallel sub-block multiplications (*PM*) slows down the application execution. On the other hand, pipelining (*P*) and flow control (*FC*) slightly improve the performance.

When we increase the number of processing nodes to eight nodes, the benefits of the

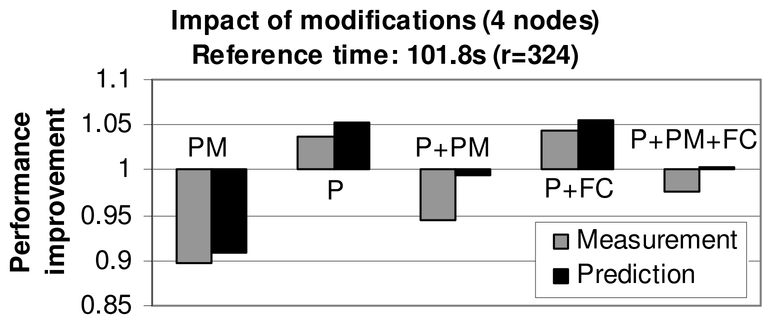


Figure 3.14: *Variation of computation time caused by parallel sub-block multiplications (*PM*), increased pipelining (*P*) and flow control (*FC*), when the matrix is split into two column blocks per node (4 nodes). Prediction errors are below 5%.*

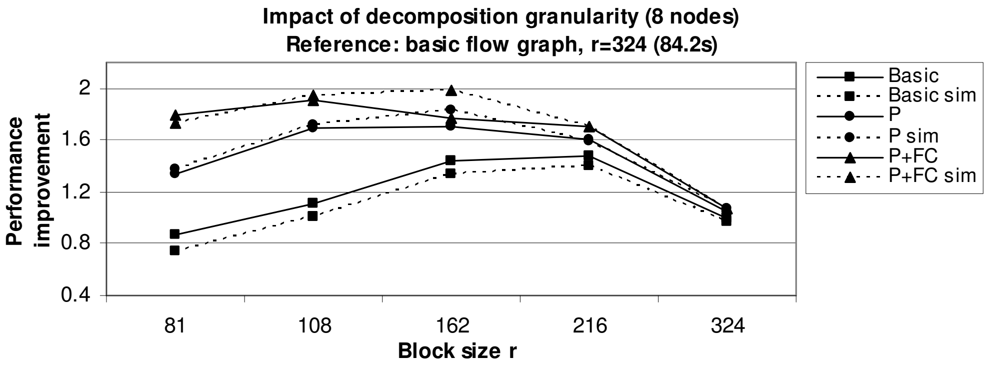


Figure 3.15: *Impact of the decomposition granularity on the performance of different pipelining strategies (8 nodes).*

pipelined flow graph (*P*) and of the flow control (*FC*) become more significant (Figure 3.15). The optimal block size for the LU factorization is also influenced by the parallelization strategy. In all cases, pipelining considerably improves the performance with respect to the basic flow graph, and the conjunction of pipelining and flow control further improves the results. Note that the change in the number of operations performed and messages sent during execution, which grows from 352 when r = 324 to about 22,000 when r = 81, has no visible impact on the prediction accuracy.

We now consider the impact of reducing the number of multiplication threads during execution. In our test case, the 2592 × 2592 matrix is split into eight column blocks distributed onto four nodes (r = 324), and the computation is performed using the basic flow graph, allowing to clearly separate the different iterations. Figure 3.16 shows the dynamic efficiency (i.e. the efficiency at each iteration step) of the application. During the first iteration, four nodes are about

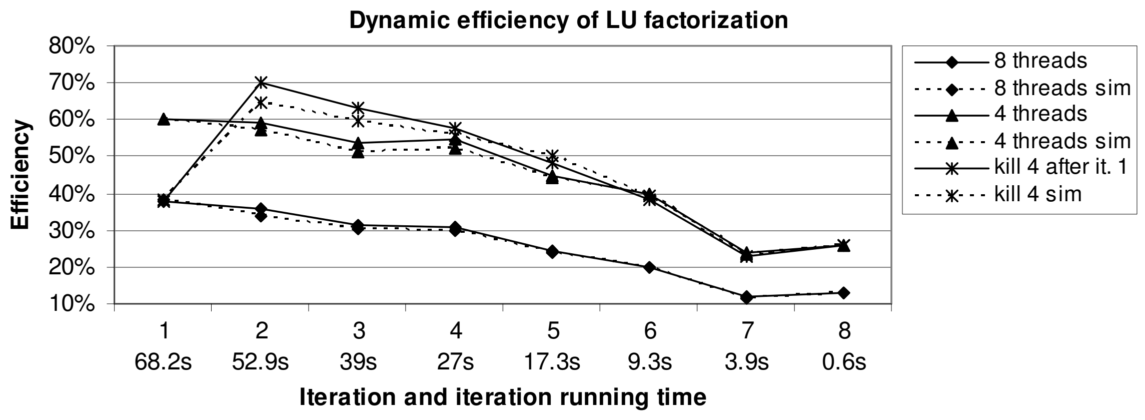


Figure 3.16: *The parallel computation of LU iterations becomes less efficient over time. Removing threads during execution increases the efficiency of the subsequent iterations.*

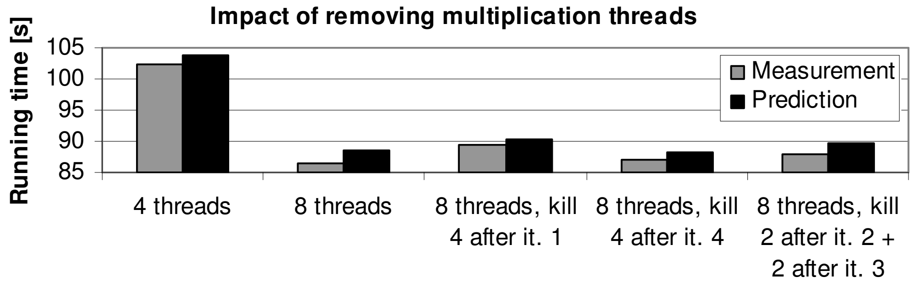


Figure 3.17: *Measured and predicted running times of different dynamic thread removal strategies.*

50% more efficient than eight nodes (60.2% vs. 37.6%). The relative efficiency of 4 nodes versus 8 nodes increases up to iteration 6 where 4 nodes have twice the efficiency of 8 nodes, i.e. iteration 6 has the same running time on 4 nodes and on 8 nodes. Therefore, removing nodes during execution should not have a large impact on the total computation time.

This is confirmed by measuring the total execution time of the application for different thread removal strategies (Figure 3.17). Using eight nodes for the whole computation or only for the first iteration yields almost the same running time, and being able to deallocate four nodes after the first iteration greatly increases the dynamic efficiency of the application (Figure 3.16, "kill 4 after iteration 1"). Figure 3.18 displays the real and simulated trace of the corresponding computations (network transfers are hidden for readability).

Since the first iteration accounts for approximately 25% of the parallel running time, the service rate of the cluster can be significantly increased if the deallocated compute nodes are

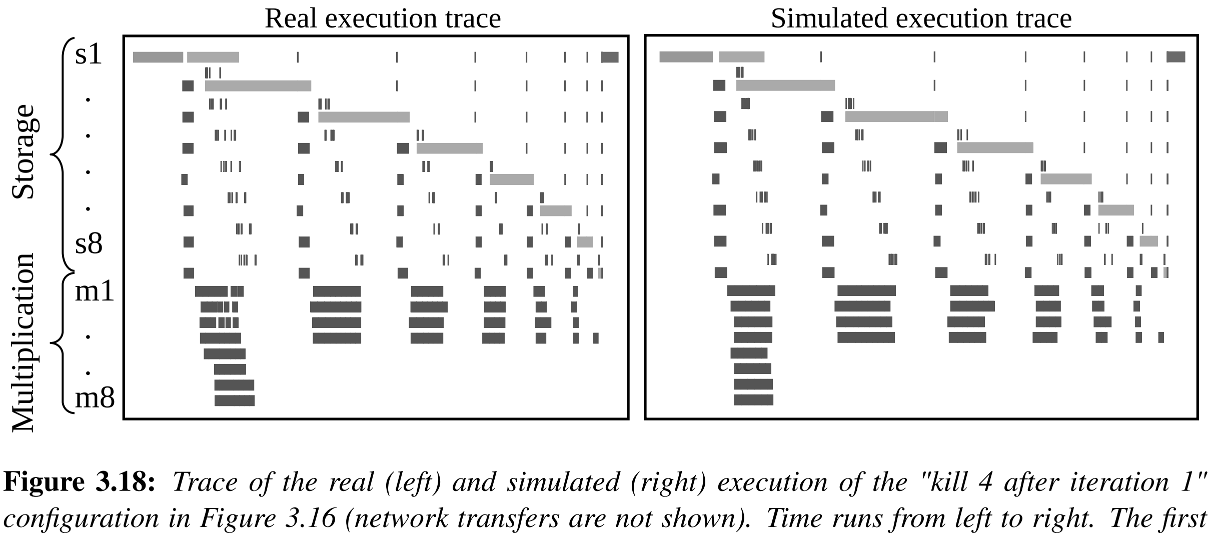


Figure 3.18: *Trace of the real (left) and simulated (right) execution of the "kill 4 after iteration 1" configuration in Figure 3.16 (network transfers are not shown). Time runs from left to right. The first eight pairs of lines represent operations running on threads s1–s8 that store column blocks. The last eight lines represent operations on multiplication threads m1–m8 (dark gray), four of which are removed after the first iteration. All the other threads run on the four remaining compute nodes.*

assigned to other applications. In this example, the execution with the static node allocation uses eight nodes during 86.9 seconds, or 695.2 seconds of total CPU utilization. The dynamic allocation strategy requires eight nodes during 22.5 seconds and four nodes during 66 seconds. The total processor utilization is therefore reduced by 37% to 438.4 seconds. Both the real and simulated executions lead to the same conclusion.

3.6 Second test application: Traveling Salesman

The second application we simulate is a simple parallel solver of the traveling salesman problem. Given a set of cities and the distances between them, the goal is to find the shortest tour enabling the salesman to visit every city. Our implementation uses a branch-and-bound algorithm with a depth-first tree traversal. Paths are constructed by visiting nearest neighbors first so that relatively good solutions are found rapidly. The algorithm stops the exploration of a subtree as soon as the path is longer than the current best solution. Since good solutions enable paths to be pruned, finding good solutions early enable a more aggressive pruning of the search tree. The efficiency of the pruning, and by extension the performance of the application, is therefore data dependent.

When tasks are distributed on several compute nodes, the ones that find good solutions quickly perform less work. In order to increase the application speedup, the globally optimal solution should periodically be distributed onto all the nodes. Since the time required to search one set of paths is highly variable and unknown *a priori*, load balancing strategies also need to be applied. Achievable speedups are therefore highly dependent on the distribution of good solutions within the input dataset. The present test does not try to optimize the parallelization of the traveling salesman problem, but rather to determine the accuracy of the speedups predicted by the simulator.

3.6.1 Implementation and validation

The parallelization of the application is described using a simple split-leaf-merge flow graph that is shown in Figure 3.19. The split operation distributes tasks specifying parts of the search space to leaf operations running on the compute nodes. In order to speed up the execution of subsequent operations, each thread keeps a copy of the best path found locally. The merge operation stores the best solution found so far within its local thread storage. In the basic implementation, tasks are distributed to nodes in a round robin fashion. Some nodes may therefore finish their tasks earlier and remain idle.

The fact that each thread only has access to local rather than the global best paths also reduces the performance of the heuristic by preventing pruning solutions that are globally suboptimal. In a second variant, we use the built-in flow control (Section 2.6.2) and load-balancing (Section 2.6.3) capabilities of DPS to equilibrate the load. Running the *SendPrefix* and the *MergePaths* operations on the same thread enables the split to read the current best solution written by the merge. The split may thus include the best solution received by the merge within its output messages, thereby periodically distributing the current best global solution to all threads.

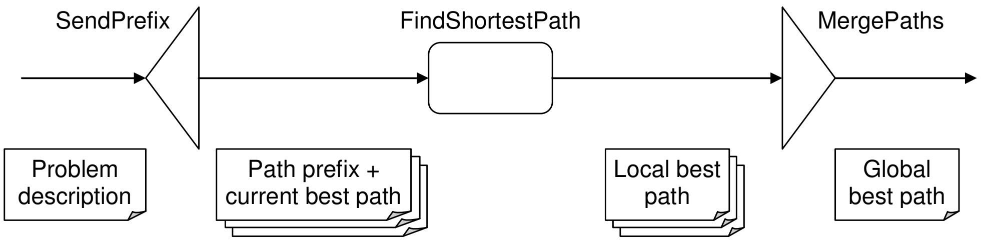


Figure 3.19: *Flowgraph solving the traveling salesman problem. We activate the flow control between the split and the merge operations to distribute the current best solution to the compute threads and increase the efficiency of the heuristic.*

We ran measurements and simulations for two problem sizes on a cluster of 2.4GHz Pentium 4 nodes. The left graph of Figure 3.20 shows the measured and predicted speedup for a problem with 17 cities. The use of dynamic load balancing and the distribution of better solutions to the threads allows the application to reach a speedup of 6 on eight nodes. The distribution of solutions within the search space of the second data set comprising 23 cities is less favorable. This leads to a lower speedup, despite the larger running time of the applica-

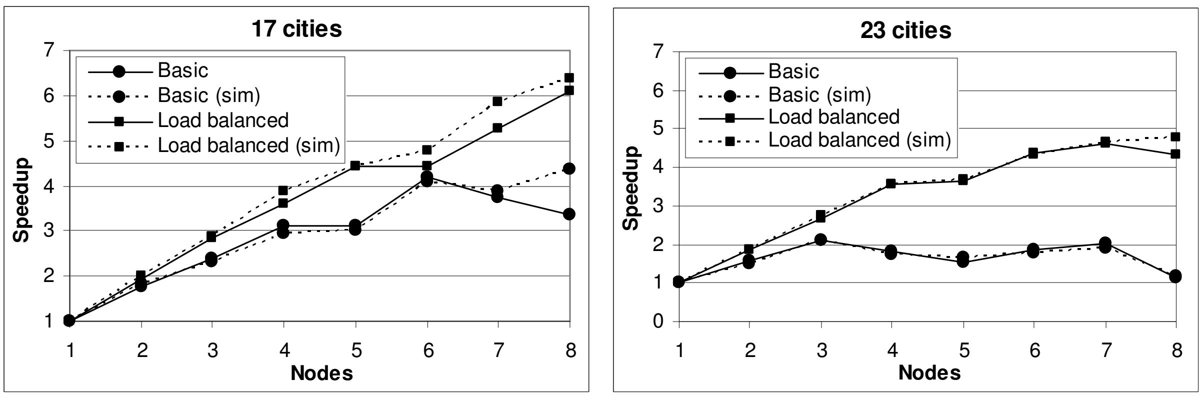


Figure 3.20: *Measured and predicted speedups for a traveling salesman solver.*

tion. Without load-balancing and periodic distribution of the current best solution to all threads, the speedup is nearly inexistent as the additional processing power either unused or wasted on searching paths that could be pruned. For all tests, the speedups predicted by simulation and the ones actually measured differ by 5.3% on average.

3.7 Improving simulation times and portability through partial direct execution

Let us now analyze to which extent the simulation time and memory use can be reduced by partial direct execution.

Out of our two test applications, it is worth noting that the TSP solver is not a good candidate for partial direct execution. The pruning heuristic indeed causes the duration of every operation to depend on the length of the best path known to the computation thread. Changing the ordering of the operations or the distribution of the current global optimum influences how fast good solutions are found, and consequently the duration of every operation. This variability makes it hard to predict running times analytically.

In contrast, the LU factorization application is very suitable for partial direct execution. The duration of each operation only depends on the size of the processed matrix and not on its content, and the time consuming functions can be easily benchmarked.

3.7.1 Preparing the LU application

We implement partial direct execution (PDEXEC) by replacing calls to the matrix multiplication, *LU*, *trsm*, and row flipping functions with simulator notifications incorporating the corresponding benchmarked times.

We then perform additional modifications to prevent the allocation of the matrices (NOALLOC), which represent most of the memory consumption. Matrices are serializable objects, which must therefore be modified to allow the internal memory buffer to be unallocated. This consists in adding a *simResize* method to initialize the variables storing the matrix size without performing the actual allocation, and in adding checks to prevent crashes in other methods that assume that the data is allocated, such as methods that print or zero the matrix content.

Appendix A illustrates the modifications applied to the operations (a) and (d) of the LU factorization flow graph (Figure 3.9).

3.7.2 Results

Table 3.1 displays the time required to simulate the execution of the LU factorization of a 2592 $\times$ 2592 matrix, with the real application running on eight nodes, using the basic flow graph and the decomposition granularity r = 216. For reference, the real parallel execution lasts 62.3s, and the real serial execution lasts 185.1s. With a running time of 193s, the overhead of the simulator when direct execution is used is 4.3%. When we use partial direct execution and we remove matrix allocations (PDEXEC NOALLOC), the simulation is almost ten times faster than the parallel execution on the same hardware and it consumes only 14MB of memory. The predicted running time changes by only -1.3% compared with the direct execution simulation.

Table 3.1: *Comparison of simulation times and memory consumptions in different simulation settings, and corresponding predicted running time. The real application running time is 62.3 s (in bold).*

	Running time [s]	Memory usage [MB]	Predicted running time [s]
UltraSparc II 440Mhz (Solaris)			
Real application (8 nodes)	**62.3**		
Real application (1 node)	185.1	108	
Direct execution (sim)	193.0	127	60.7
PDEXEC (sim)	9.1	124	60.3
PDEXEC NOALLOC (sim)	6.5	14	59.9
Pentium 4 2.4GHz (Windows)			
Direct execution (sim)	29.7	127	
PDEXEC (sim)	2.5	124	60.0
PDEXEC NOALLOC (sim)	1.6	14	59.9

We actually used this optimized simulator mode to produce all the simulation results presented in Section 3.5.3. Its prediction accuracy for the 168 measurements carried out for establishing the results are shown in Figure 3.21. 71.4% of all predictions are within $\pm$4% accuracy, 81.6% are within $\pm$6% accuracy, and more than 95% are within $\pm$12% prediction accuracy.

Table 3.1 displays simulation results for two platforms with different CPU speed, CPU architecture and operating system. Since the Pentium 4 processor is much faster than the UltraSparc II, prediction results based on direct execution are not representative. However, when partial direct execution is used, the faster processor has nearly no impact on the predicted running time of the LU factorization application. In order to assess the portability of our simulator, we ran a same set of simulations on three systems with single processors at 600MHz, 2.4GHz and 3GHz, and on one system with two dual-core 2.6GHz processors. The simulation

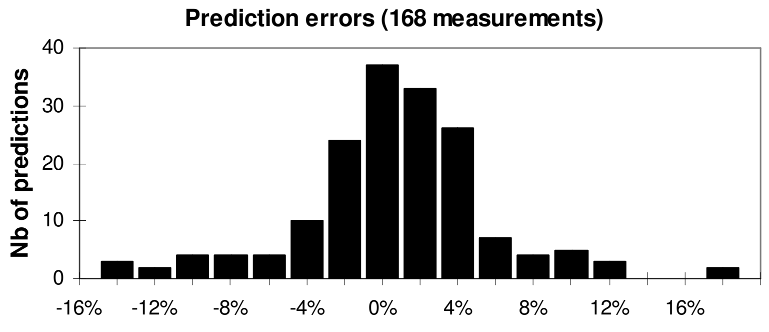


Figure 3.21: *Histogram of prediction errors with partial direct execution.*

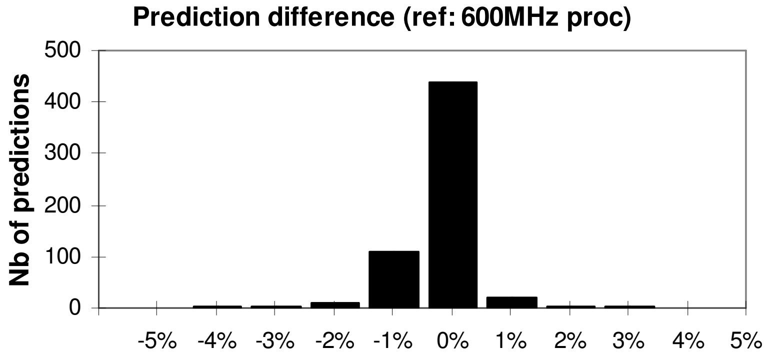


Figure 3.22: *Histogram of relative differences of running times predicted on a 2.4GHz, a 3Ghz and a quad-2.6Ghz systems with respect to predictions performed on a 600MHz system (3 times 200 comparisons in total).*

set consists of 100 different application configurations, combining the variants presented in Section 3.5.2, namely, different number of nodes, decomposition block sizes, the use of flow control and parallel sub-block multiplications. We ran all 100 simulations with and without matrix allocation (NOALLOC), producing 200 prediction results. Figure 3.22 shows the relative difference of the 200 predictions produced by each one of the faster 2.4GHz, 3Ghz and quad-2.6Ghz systems, compared to the predictions produced by the slower 600MHz system (600 comparisons in total). Despite the performance difference, 97% of the prediction results differ by only $\pm 2\%$. The outliers with an error greater than 5% represent 1.3% of all measurements. The fact that predictions made on the multiprocessor system match results obtained on single processor systems shows that the execution of the various computation threads is correctly sequenced.

There is a little skew in the histogram, as the number of predictions differences between -1.5% and -0.5% is much larger than the ones between $+0.5\%$ and $+1.5\%$. This is explained by the fact that the simulator is always measuring the duration of atomic steps. While the most expensive computations are taken into account thanks to the benchmarked values, the code that must still be executed is sped up on the faster systems. This slightly reduces the overall duration of each atomic step, and introduces a bias in the predictions. Nevertheless, the small magnitude of this effect indicates that all the significant contributions have been accurately benchmarked.

3.8 Simulations for application analysis

As described in the previous section, modeling the duration of the individual operations and message transfers of a DPS application decreases the running time and memory consumption of the simulated application. It also leads to a parametric model of the application [58]. Since parametric models allow the different performance factors to be isolated from one another, they enable analyzing the sensitivity of the overall running time with respect to each individual parameter. Varying the running time of specific operations helps identifying the operations located on the critical path of the computation and quantifying the potential benefits of their optimization.

Hardware parameters (bandwidth, latency, CPU for communications) can be modified directly within the cluster configuration file. Operation durations can be modified by changing the prediction models e.g. by adding a scaling factor to simulate a faster or slower CPU [18, 39, 57], or in our case by changing the durations collected in the benchmark file.

For both the basic and the pipelined flow-controlled LU factorization flow graphs, we simulated a high performance network by reducing the latency and increasing the bandwidth parameters in the cluster parameterization file (the chosen numbers are representative of currently achievable performance [15, 71, 90]). In order to study the improvement brought by the overlap of communications and computations under various conditions, we also reduced the CPU utilization for the communications. Our first simulations consider eight compute nodes and a coarse decomposition with one column block per node ($r = 324$). The results are summarized in Table 3.2. Such a decomposition produces fairly large messages and the latency parameter contributes little to their transfer time (line 2), while the bandwidth parameter plays a more important role in the total application running time (line 3). Due to the better overlapping of computations and communications provided by the pipelined flow graph, communication times are partly hidden. Therefore the performance increase brought by the improved network parameters is lower than for the basic flow graph. The factorization of the blocks on the matrix diagonal (operation (f) in Figure 3.9) lies on the critical path of the execution for the basic flow graph. Speeding up the LU computations by 10% reduces the overall running time by the same duration (4.1 seconds) for both parallelization strategies (Table 3.2, last line).

Table 3.2: *Predicted running times with one column block per node on eight nodes (r=324), for varying application and cluster parameters. The relative difference with respect to the predicted running time with the original parameters (in bold) is displayed next to every prediction. The original network is Fast Ethernet, with a latency of 1350μs and a bandwidth of 11.85MB/s.*

r=324	Basic flow graph		Pipelined flow graph + flow control	
	Predicted running time [s]	Relative difference in respect to original parameters	Predicted running time [s]	Relative difference in respect to original parameters
Original parameters	**86.5**		**78.3**	
Latency=2μs	86.2	-0.3%	78.1	-0.3%
Latency=2μs Bandwidth=912 MB/s	72.7	-16.0%	69.5	-11.2%
CPU utilization for comm. divided by 4	82.9	-4.2%	75.9	-3.1%
LU computation 10% faster	82.4	-4.8%	74.2	-5.2%

Table 3.3: *Predicted running times with three column blocks per node on eight nodes (r=108), for varying application and cluster parameters. The relative difference with respect to the predicted running time with the original parameters (in bold) is displayed next to every prediction.*

r=108	Basic flow graph		Pipelined flow graph + flow control	
	Predicted running time [s]	Relative difference in respect to original parameters	Predicted running time [s]	Relative difference in respect to original parameters
Original parameters	**83.9**		**43.0**	
Latency=2μs	77.2	-8.0%	41.2	-4.1%
Latency=2μs Bandwidth=912 MB/s	30.4	-63.7%	24.9	-42.0%
CPU utilization for comm. divided by 4	83.1	-0.9%	31.2	-27.4%
LU computation 10% faster	83.2	-0.8%	42.6	-1.0%

Table 3.3 shows the same set of measurements performed when the application runs with a finer grain decomposition (3 column blocks per node, r = 108). The total amount of data transferred over the network grows by a factor of 3 (1.3 vs 0.4 GB), and the number of messages increases about 24 times (14701 vs. 613). The lower computation to communication ratio induced by smaller blocks causes network transfers to account for a greater part of the overall running time. Since messages are smaller, the network latency also becomes an important factor. Both considerations are reflected in the simulation results, where improved network parameters reduce the running time much more than with the coarser decomposition used in Table 3.2. Their impact is smaller, but remains important, for the pipelined flow graph. On the hardware used for our real execution measurements, handling multiple simultaneous transfers to eight nodes requires more than 50% CPU utilization. This factor is very important for the pipelined flow graph due to the large overlap between communications and computations. Dividing this CPU utilization for communications by four therefore significantly decreases the application running time (-27% in Table 3.3, line 4). The increased decomposition granularity reduces the weight of the LU factorization operations. Since the network is now the bottleneck, making the LU computation faster yields very little benefits for both flow graphs (Table 3.3, last line).

The impact of the excessive network utilization of our parallel LU factorization implementation is even more apparent when we simulate faster processors by dividing all computation times by four and by reducing the CPU consumed by communications by a factor of four. Table 3.4 shows results for both r = 324 and r = 108. Improving the latency and the network bandwidth now yields very significant running times reductions in all configurations. As expected, the faster processors reduce running times in all cases. However, the basic flow graph now runs faster when using the coarser and less network-intensive decomposition (one column block per node, r = 324). As expected, the pipelined flow graph performs better than the basic flow graph.

The presented results show that each one of the selected hardware parameters, i.e. the network latency, network bandwidth and the CPU consumption for communications, has a significant impact on the application running time. The quality of the predictions obtained in the previous sections show that this parameter set is sufficient for characterizing the behavior of a cluster composed of a small set of single processor computing nodes. Despite the approximations made within the models and within the simulations, our simulator can therefore be used as a performance analysis tool.

Table 3.4: *Impact of network parameters on predicted running times when the duration of all individual computations and the CPU consumption of communications have been reduced by a factor of 4.*

	Basic flow graph		Pipelined flow graph + flow control	
	Predicted running time [s]	Relative difference in respect to original parameters	Predicted running time [s]	Relative difference in respect to original parameters
$r = 324$				
4x faster processors	36.6		25.4	
Latency=2μs	36.3	-0.9%	25.3	-0.3%
Latency=2μs Bandwidth=912 MB/s	18.3	-49.9%	17.5	-31.0%
$r = 108$				
4x faster processors	76.2		24.2	
Latency=2μs	69.3	-9.0%	21.0	-13.2%
Latency=2μs Bandwidth=912 MB/s	8.3	-89.1%	6.6	-72.5%

3.8.1 Randomizing atomic steps durations

We only used deterministic atomic step durations in our predictions so far. Running multiple simulations with the same parameters therefore produces the same prediction results. In practice however, small variations do occur and they may have a surprisingly large impact on the overall application running time as they may cause events to be reordered within the application. Since incoming messages are queued within threads, changing the ordering of their delivery changes the order in which they are processed. If such reorderings postpone the execution of the computations within the critical path determining the application running time, a delay of a few milliseconds in the delivery of a message may considerably slow down the application execution (you can find an example of such a situation in Section 4.2).

The sensitivity of the overall application performance to such variations can be studied by adding a small positive or negative time variation to operation durations and network transfer times [111].

3.9 Limitations and possible improvements

Let us recall our cluster hardware parameterization and execution model. The three parameters describing the hardware are the network latency, the peak network bandwidth and the CPU usage consumed by network transfers as a function of the number of simultaneous transfers. Considering the cluster architecture, we assume that all nodes are identical and have a single processor, and that they are interconnected through a single, infinite-bandwidth crossbar switch via full duplex links. This section describes the limits of this parameterization and of the cluster model, and describes potential future work.

3.9.1 Internal DPS delays

The current model neglects the time required to serialize and deserialize messages. Serialization costs increase significantly when complex data structures such as vectors or trees are serialized (Appendix B). Secondly, we ignore internal latencies within the DPS runtime system. Although these latencies have been measured to be less than 100μs per operation [35], the accuracy of predictions is likely to decrease for fine-grain applications performing many very short operations and sending many very small messages. Startup up costs, such as the time needed to start up remote instances, thread collections and threads are also not taken into account. Since they occur only once, their impact decreases as the application running time increases.

Both the internal DPS latencies and serialization overheads are partly taken into account if we parameterize the network using the tool described in Appendix A, which measures the latency and bandwidth parameters by sending simple serializable objects of different sizes within a regular DPS flow graph. Measurements therefore include the time spent by messages within the DPS runtime before being serialized and sent, and the time needed to create an operation and deliver the message to its *execute* method.

3.9.2 Clusters of heterogeneous computers

A heterogeneous cluster refers to a network of workstations bringing together machines with different performance, as well as different CPU architectures and operating systems. DPS applications may run on such clusters thanks to the fact that the serialization mechanism is able to transfer data between nodes with different byte-ordering and pointer sizes.

The DPS simulator can fully support such clusters thanks to partial direct execution. The application developer must however provide running time predictions for each machine type. An alternative is to use a set of reference running times, and to use scaling factors to adapt those times to the various machines types composing the cluster [18, 39, 57].

3.9.3 Multicores and accelerators

New trends in CPU development will require changes in the way the simulator operates. The first trend is already well established and relates to multicore systems. While two threads running on a single CPU are interleaved with a minimal performance penalty (typically less than 1%), two threads running on multiple cores or processors have to share the available memory bandwidth. This competition reduces the speed at which data reaches the processing units and slows down memory-intensive applications. In the worst cases, the multi-core running time may be larger than the single core running time as memory contention increases the latency experienced when fetching data from the main memory. The actual performance impact depends both on the particular computations being performed and on the CPU and memory architectures running the computations [43].

The second trend is the dynamic adaptation of CPU frequencies to the needs of the applications. For instance, Intel's Core 2 Duo mobile processors include the *Intel Dynamic Acceleration Technology*. When one core is idle, the processor may provide more electrical power to the other core. Single-threaded applications therefore get a performance boost, while the temperature of the processor remains within thermal dissipation constraints. The impact of both the underlying memory architectures and dynamic frequency adaptation are evaluated in [57, 104]. While such techniques are already being used in laptops in order to extend their battery life, they have not yet been used in processors dedicated to desktops and servers.

In respect to our simulator, these developments require some adaptations to our model. Varying CPU frequencies can be represented by adapting the timeslice removed from atomic steps at every iteration: if a CPU becomes faster, the timeslices allocated to the atomic steps that it runs are increased by a factor proportional to the change in frequency. Assuming that DPS operations only perform serial computations, a compute node with n_{proc} processors or cores could be modeled by allocating a fraction of the timeslice equal to $\max\{1, n_{\text{proc}}/n_{\text{ops}}\}$ to each of the n_{ops} operations executing simultaneously. Such a model fails to take into account memory contention however.

Recent developments also show specialized pieces of hardware being used to accelerate specific functions. The most popular of these accelerators are probably graphics cards used for general-purpose computations. For instance, CUDA, a development environment for NVidia GPUs triggered many developments ranging from DNA sequence alignment to linear algebra [10, 75, 83]. The Cell Broadband Engine processor from IBM found in Playstation 3 game consoles is a small parallel machine on its own, composed of a Power processor and eight Synergistic Processing Elements (SPE) co-processors communicating over a single high-throughput ring network. It shows impressive peak performance, and several algorithms have already been ported [11, 62]. Finally, in embedded systems, some research looks into the use of FPGAs as customized hardware for accelerating time-consuming functions [13].

Simulating such accelerators is easy as long as they can be considered as a dedicated black box performing serial computations. If that is the case, the existence of the accelerator simply translates into shorter computation times. Problems arise when for instance multiple threads call a single GPU or Cell accelerated function simultaneously. The simulator must know whether the calls are sequenced or executed simultaneously. In the latter case, resource sharing schemes that take into account the specific architecture within the accelerator must be used.

3.9.4 Network topologies

Our networking model assumes that all nodes are connected through a single infinite-bandwidth network switch and that no contention exists on the network.

The use of a single switch is common for clusters with less than a few hundred nodes[1]. The internal bandwidth of these switches is generally proportional to the number of ports. The low node count used in our validations certainly did not hit any limit within the switch. Larger scale clusters, grids (in the sense of clusters of geographically dispersed nodes), and integrated supercomputers such as the Blue Gene/L have more sophisticated network topologies such as fat trees or 3D-torus that we do not currently simulate.

Our assumption that no contention exists within network links underestimates the cost of communications. The simulator may therefore favor communication intensive application configurations that are suboptimal in a real setting. The fact that our implementation of the LU factorization is fairly network intensive (Section 3.8) did not prevent the simulator from producing accurate predictions in our tests. However, contention is likely to have larger effects when more nodes are involved.

[1]It is the case for the Pleiades cluster as well as for the ones we used to validate our simulator

3.9.5 Multi-application simulations

The simulator supports the ability of simulating multiple applications simultaneously. When started, each application creates its own controllers, thread collections and operations. The simulator then takes care of distributing the CPU and network resources among all applications. Different threads share CPU resources in the same way if they belong to the same thread collection than if they belong to different applications.

Application startup times are read from an external scheduling file. Each line contains the name of the application class (e.g. *MergeSortApp* in Listing 2.6), followed by a timestamp indicating the starting time of the execution, and by the command line parameters that would be passed to the application in a regular execution. The timestamp corresponds to the startup time according to the simulation clock. The command line parameters may be used for instance to specify the original mapping of threads onto processing nodes. Whenever the simulator clock reaches one of the timestamps, the simulator instantiates the associated application and calls its *start* method.

The current implementation requires that all applications are compiled into a single executable. The use of global variables and constants and the naming of classes and methods may thus need to be adapted. The multi-application simulator provides its own *main* function which initializes the DPS runtime and the simulator.

Multi-applications simulations can be used to study the potential benefits that can be drawn from sharing compute nodes between applications. However, we did not validate the current execution model against actual applications. Moreover, exploiting the full potential of such simulations would require the availability of a dynamic runtime scheduler. Such a scheduler could then use cluster utilization information and leverage the malleability features of DPS to migrate, add and remove application threads to minimize application running times, and maximize the cluster utilization.

3.10 Conclusions and future work

The performance of a parallel application not only depends on its implementation, but also on decomposition parameters, on tasks to nodes mapping and on node allocation decisions. The choice of optimal parameters may depend on the CPU and network speed, as well as on the application input data. The dynamic allocation of compute nodes during the execution of parallel applications can further improve the utilization of cluster resources. In order to help decide how and when the allocation should be modified, we introduce the concept of dynamic efficiency which expresses the resource utilization efficiency as a function of time. We obtain information about the performance and the dynamic efficiency of parallel programs by running a simulator on top of the parallelization framework runtime system.

In the presently used Dynamic Parallel Schedules framework, computations are performed by threads, which can be dynamically allocated or deallocated onto compute nodes. We simulate the parallel execution of an application by running all threads within a single application instance. The simulator then coordinates and synchronizes the execution of the threads to control the application execution. Communication patterns, as well as the number of messages and operations are derived through direct execution.

By default, the duration of each operation is also obtained through direct execution. The running time, memory requirements and portability of the simulation are improved by using partial direct execution, i.e. by replacing time-consuming computations with running time predictions, and by avoiding large memory allocations. Varying the duration of individual operations enables determining the operations that belong to the critical path and that can benefit from further optimizations.

We describe a simple model for typical cluster configurations that accurately takes bounded and shared network and CPU resources into account. We verify the prediction accuracy of our simulator by applying several parallelization and deployment strategies to a pipelined LU factorization application and to a load-balanced traveling salesman problem solver. The LU factorization application also shows that the simulator is able to accurately predict running times and dynamic efficiency when deallocating compute nodes at different time points of the program execution. By varying the simulated hardware parameters such as the processing power of the compute nodes, the network latency and bandwidth, and the CPU utilization of network communications, we identify the performance bottlenecks within the application and validate our cluster parameterization and resource sharing model.

The integration of the simulator within the parallelization framework allows applications to be simulated through a single recompilation. Simulations may produce detailed execution statistics, as well as execution traces that can be visualized like the traces of regular executions. A cluster parameterization tool is provided to easily obtain the required hardware parameters. Benchmarking utilities facilitate the implementation of partial direct execution within the application. The subsequent ability to vary the duration of individual operations may help deepen the understanding of the application behavior and focus further optimization efforts.

Although results are presented here in the context of DPS, the resource sharing model and the partial direct execution principles of the simulator can be adapted to other parallelization frameworks or libraries. The results presented in this chapter have been published in [96, 100].

Chapter 4

Message Race and Deadlock Detection

4.1 Introduction

Parallel applications are vulnerable to types of errors to which single-threaded applications are immune, that stem from the non-deterministic orderings of events within the application. We focus here on deadlocks, when conflicts over resources prevent the application from moving forward, and on message races, when changing the order of delivery of messages in a message-passing parallel application changes the result of the computation.

Adding synchronizations between the participating processes or threads makes executions more predictable. It gives developers control over the program execution by reducing the amount by which slow and fast processes may drift apart. However, as we saw in the previous chapter, synchronizing threads or processes is generally detrimental to performance. One of the major difficulties when developing a parallel program is therefore to simultaneously ensure that an application has good performance and that it produces correct results independently of its execution environment.

Synchronization errors are difficult to detect because they are intermittent. Removing or loosening a synchronization actually required for the correct execution of an application often does not immediately translate into erroneous results. Errors may for example occur only once every hundred runs, or only when more than ten threads are used, or only when one compute node is much slower than the others. They may remain hidden until new hardware is bought. Failing to detect bugs early may also have psychological effects: as developers build up confidence in their implementation over time, results become less and less likely to be challenged.

Tools able to provide guarantees about the correctness of the results produced by an application are therefore very valuable. Their first benefit is that they may encourage developers to write applications with more efficient synchronization patterns. However, they may provide debugging and understanding aids by revealing errors, and by enabling such errors to be reproduced systematically.

This chapter only considers issues related to the *parallel* implementation of an application, and focuses on detecting executions that produce different computation results. Errors made before or independently of the parallelization phase, for instance during the translation of a mathematical problem formulation into its computer implementation, are therefore out of the scope of this research. We also note that the methods presented here currently assume that the number of threads does not change during the application execution.

4.1.1 Underlying concepts and assumptions

The basic idea developed throughout this chapter is to execute an application for all possible orderings of messages. Finding an ordering that produces a different outcome then reveals a message race. If all orderings produce the same final results, we have a guarantee that no message race exists in the application, for the particular input that has been tested[1]. These statements are valid provided that two conditions are met. Firstly, threads must exchange information only via messages, i.e. there is no simultaneous access to non-read-only shared memory. Secondly, computations must be deterministic. Two executions where messages are delivered in the same order therefore produce the same messages and the same computation results. In other words, the only non-determinism lies in the ordering in which messages are delivered.

These assumptions are reasonable in practice. The use of non-constant shared variables is discouraged by the execution model of DPS, where threads should be allowed to migrate and computation results should not depend on the mapping of threads onto processes. As for our second requirement, developers often remove non-determinism from their application during debugging in order to have reproducible results.

Unfortunately, the exponential number of possible message orderings makes it impossible to simply execute them all and compare the final computation result after each run. Reducing the number of orderings to be tested is therefore a key requirement for any method to be useful

[1]The automated tests described in this chapter actually consider the full application state rather than only its output. Message races that have no effect on the output are therefore also detected (Section 4.6).

in practice.

We start the present chapter by illustrating how message races and deadlocks may occur within DPS applications (Section 4.2). We then show how synchronization errors can be detected using three different methods. We reuse a simple example from Section 4.2 in order to illustrate our various approaches.

Our first approach relies solely on the knowledge and intuition of the developer. Section 4.3 focuses on manually testing interesting orderings using a graphical debugger for DPS applications. Thanks to the flow graph, the state of application executions can be represented visually in a very natural way. Various types of breakpoints enable the developer to suspend and resume the execution of individual computation threads and to change the ordering of events. The ordering of computations can be further influenced by reordering messages that await processing within reception queues. Messages can be altered from within the debugger in order to change the application behavior. Execution scenarios that occur only rarely in actual executions can thereby be explicitly tested. In all cases, the graphical representation provides both a high level and detailed instantaneous view that facilitates following and understanding the application execution.

The next two sections consider the automatic generation and execution of possible message orderings. Section 4.4 starts by considering the subset of applications that generate a *fixed message set*. For these applications, the number, content and destination of the messages produced during the program execution are independent of the order in which messages are delivered. If the set of messages produced is fixed, the communication pattern and the location of computations are fixed as well. The causality between events is therefore fixed and can be captured using a Partial Order Execution Graph (POEG) [20]. Our first method performs a static analysis of the graph and identifies graph partitions containing independent sets of messages. Orderings of messages belonging to different sets may then be re-executed independently from each other. Within each set, equivalent orderings are identified and eliminated using a partial-order reduction technique that leverages knowledge about whether computations read or modify the local memory. Reducing the number of equivalent orderings ensures that only relevant cases are tested, therefore increasing the likelihood that existing message races are revealed. Moreover, the decomposition of the application execution allows the developer to focus his effort on specific parts of the application that may be difficult to debug. Once a race is detected, the decomposition isolates its potential sources within the part being tested.

This static decomposition can be applied to a wide range of problems. For instance, most linear algebra computations and finite elements methods have the required property of producing a set of messages that does not depend on the order in which they are processed by the application. In contrast, applications that incorporate load-balancing mechanisms or heuristics taking advantage of previous computation results do not produce fixed sets of messages. Branch and bound optimization problems that use the current best solution to prune the search tree belong to the latter category.

Our second method, described in Section 4.5, presents a more general approach which removes the fixed message set requirement. We dynamically construct and explore a *message-passing state graph* that greatly reduces the cost of testing possible orderings. We identify application states common to multiple orderings by comparing checkpoints taken after the delivery of every message. This ensures that each state appears only once in the state graph, such that sequences of computations common to multiple orderings are not needlessly re-executed. We then use information about communication patterns and read-write accesses to local process variables to reduce the number of explored states. This approach greatly reduces the replay time at the expense of the memory or disk space needed to store intermediate application checkpoints. In order to handle cases where the space and time requirements are too large, we also describe an algorithm that tests a subset of orderings that has a high probability of revealing commonly found message races.

We implemented the two automated race and deadlock detection techniques on top of the DPS simulator. The testing procedure is therefore carried out within a single multithreaded process. We present results for four different parallel applications. It is sufficient to recompile a DPS application in order to activate the message race and deadlock detection mechanisms. Any modification to the application code or input data can therefore be immediately tested. Detected erroneous executions can then be replayed [20, 48] for debugging purposes.

The remaining sections compares the results obtained with the static POEG decomposition and the dynamic message-passing state graph building techniques, and show how their combination brings further improvements. Future developments that combine manual and automated testing are also sketched. In this Chapter, the ideas and example applications are presented in the context of DPS applications. They can however be generalized to other message-passing models such as MPI, as we will see in Chapter 5.

4.1.2 Related work

A parallel application debugger must specifically address two sets of issues that are unknown to regular serial debuggers. The first set of issues stems from the non-deterministic ordering of events, whereas the second one lies in the overwhelming amount of information that must be filtered, aggregated and preprocessed before being delivered to the developer.

The recent advent of tera- and now peta-scale parallel systems only exacerbates the second problem. In that respect, several contributions discuss the importance of using multiple abstraction levels for parallel program debugging [60, 61, 67]. They describe interesting parallel debugging concepts, such as process isolation, time-process communication graphs and call graph representations of the underlying parallel program execution. Some debugging tools explicitly target large-scale systems. One tool [9] focuses on aggregating the textual output of the different processes and another tool [6] aggregates their stack trace to identify processes which, despite being identical, behave differently. In their case study, the authors of [6] use their tool in order to identify a subset of processes that have an erroneous behavior, and then use a distinct full featured debugger to further analyze these processes.

Multiple full-featured interactive parallel debuggers have been described in the literature, e.g. Mantis [74], TotalView [114] and p2d2 [49]. All support the isolation of specific processes, as well as attaching a sequential debugger to remote application instances, which enables breakpoints to be set in individual processes. TotalView also supports the inspection of message queues in MPI programs [23, 110]. Message queue inspection is also available in the debugger for the Charm++ parallel application development framework [53]. In the latter case, the integration with the Charm++ parallel runtime enables higher-level features such as setting breakpoints on remote entry points [55].

While these tools provide the developer with detailed information about the application execution, none of them provides an instantaneous high-level picture of its current state. More importantly, none of them focuses on the first set of issues faced by parallel applications, namely the non-deterministic orderings of events within the application. While the detection and debugging of message races received much attention from researchers, a lot of work focuses on record and replay techniques to enable reproducing a race once it has been detected [20, 48, 66, 116, 128]. Few proposals explicitly test different message delivery orderings, and to our knowledge none of them has been integrated within a debugger [59, 121].

Regarding the detection of errors, two proposals detect message races by evaluating predefined predicates, i.e. boolean-valued functions, that consider both the local and the global state of the application at various points of the execution [80, 84]. Since no control is applied on the program execution, the detection can only work for executions where message races actually occur.

Several variants of controlled re-execution of message-passing applications have been described in the literature. Mittal and Garg [78] determine where to add synchronizations in order to maintain a global predicate, thereby pointing to the location of synchronization bugs. However, they do not allow events to be reordered on a given process. Duesterwald et al. [27] describe a slicing method to isolate only problematic statements when an erroneous result is observed. The slice may then be re-executed for identifying the source of error. Kilgore and Chase [59] identify sets of messages that can be received in any order on a given process, and propose an algorithm that generates a single ordering that maximizes the number of reversed message pairs compared to the original execution. In essence, this is similar to our static decomposition method (Section 4.4), but no results are shown that would allow us to compare the two approaches.

A large body of work focuses on the analysis and verification of *shared memory* parallel applications. Since the number of possible thread interleavings is even more intractable than the number of message permutations in distributed memory message-passing programs, a central goal is the detection of equivalent interleavings. Several researchers therefore use information about read/write access to shared variables to determine the commutativity of operations and reduce multiple equivalent executions to the same serialized execution [17, 32, 125]; to the best of our knowledge, no prior work leverages information about how computations access the local memory in the context of message-passing distributed memory applications. ConTest [30] is a tool for testing multithreaded Java programs. One of its heuristics, called *halt-one-thread*, works by suspending the execution of a single thread until no other thread can continue further. We use a similar idea in both the debugger and the automated testing approaches.

Finally, several authors argue that detecting the first message race is beneficial [44, 81]. Correcting early races not only removes subsequent instances of the same race, but also prevents potential spurious races from being enabled (e.g. a race exists because a prior race invalidates some assumption made on the data). Our static decomposition method tests application parts in chronological order of their execution. In the message-passing state graph approach, we can determine at which point two executions start to diverge. We can therefore determine the temporal location of every race, thereby allowing the developer to correct early races first.

A few research groups work specifically on MPI applications. JitterBug [124] is a tool from Lawrence Livermore National Labs that increases the coverage achieved by test runs. It adds delays to MPI message transfers to influence the delivery order of messages. Siegel and Avrunin have been working on the development of formal models of MPI applications [105, 106, 107, 108], which can then be verified using a model checker. Very recently, Vakkalanka et al. presented ISP [121], a tool that automatically executes all possible interleavings of MPI applications, using partial-order reduction methods to avoid testing equivalent executions. However, unlike our automated testing techniques it implements no checkpointing, and therefore requires reexecuting the complete application for every interleaving.

4.2 Synchronization errors in DPS applications

As described in Section 2.5, the execution of DPS applications is fully asynchronous. Given the acyclic nature of the flow graph, an associated message-passing graph is therefore deadlock-free, provided that no operation terminates without outputting a message.

Message races on the other hand may occur if the execution ordering of two non-commutative operations is not constrained by the flow graph. Similarly, merge or stream operations should be implemented such that their behavior does not depend on the ordering of incoming messages. Thanks to the asynchronous execution model of DPS, deadlocks may only appear as a consequence of a message race. For instance, a buggy stream operation that fails to send any output message for some ordering of inputs prevents the execution of subsequent operations and the termination of the flow graph.

We illustrate a possible message race within an iterative neighborhood-dependent parallel application. A data domain, composed of a grid of cells, is distributed onto multiple processing nodes. At every iteration, we update the value of each cell as a function of the value of its neighbors. A node storing one domain part must then get the new value of the border cells of its neighbors in order to compute the next iteration correctly. Finite element and cellular automata computations are two application examples that can be parallelized using these principles.

Figure 4.2 displays the unfolded flow graph of one iteration of the computation. The application uses two thread collections; *main* runs the global split and merge operations synchronizing the successive iterations, and *proc* stores the pieces of the processed data domain and

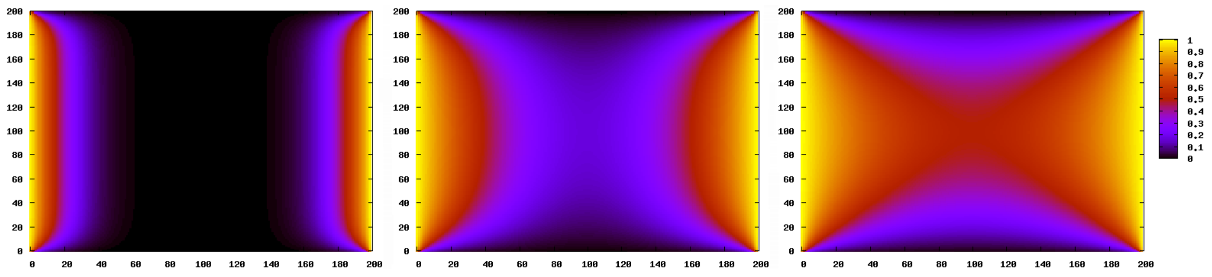


Figure 4.1: *Early, intermediate and late state of an iterative computation of the propagation of heat in a 2D rectangular domain.*

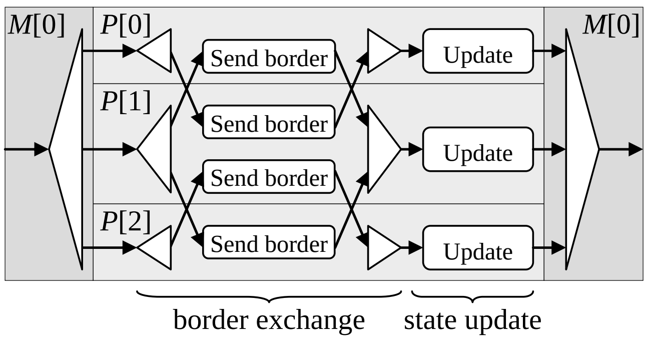


Figure 4.2: *The unfolded flow graph of one iteration of a neighborhood dependent parallel computation.*

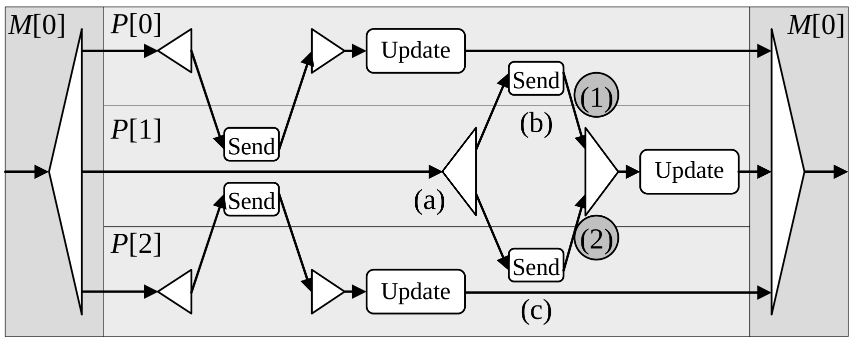


Figure 4.3: *If the processing of the split operation (a) on* P*[1] is delayed, the state of* P*[0] and* P*[2] is read by (b) and (c) after having been updated.*

compute the new state at every iteration. Here, the threads *P*[0], *P*[1] and *P*[2] belong to the thread collection *proc* and each thread stores one third of the processed data domain. At each iteration, every thread sends a request to its neighbors, which send back a copy of their subdomain border (*Send border* operation). The computation of the new state of the subdomain (*Update* operation) is performed once the requested borders have been received.

However, this flow graph enforces no synchronization on a given process between the "border exchange" and "state update" phases. Therefore, delaying some messages may have unexpected consequences. In the execution depicted in Figure 4.3, the borders sent in messages (1) and (2) have already been updated, causing incorrect values to be used to update the subdomain stored on *P*[1] and distorting the results of the computation. The existence of the race depends on the implementation of the operations: in the present case, it is nonexistent if the borders to be exchanged are stored in double buffers, allowing a copy of the old border to be kept when *P*[0] and *P*[2] perform the update. Sending the copy of the old subdomain borders in messages (1) and (2) then allows the correct computation to be performed on P[1]. Detecting the race therefore requires running the code in both orderings.

4.3 Interactive testing of DPS applications

As we can see from the multiple figures used throughout this thesis, DPS flow graphs have a straightforward graphical representation. Having the ability to visualize their evolution as they unfold during the execution of an application therefore provides an instantaneous and powerful help for understanding the application behavior. Combining such visual feedback with means of controlling application executions leads to an interactive testing tool for DPS applications.

The present section describes this tool. Manual control over the execution is enabled by an independent debugger which displays information about DPS messages, threads and operations. It is implemented as a standalone Java program to which the parallel application connects upon startup. Hooks (enabled at compilation time) within the DPS runtime send notifications to the debugger. Any DPS application thereby automatically benefits from the debugging functionality without requiring any modification.

4.3.1 Interaction between the debugger and the parallel application

When the debugging hooks are enabled within the DPS library, an extra parameter added to the application command line identifies the host running the debugger. All instances then open a TCP connection to the debugger upon startup. The application communicates both structural and runtime information to the debugger. The structural information comprises the thread collections and the flow graph used by the application, and is transferred to the debugger upon creation. During its execution, the application then generates a *sendMsg* notification for every message it sends. The notification contains a copy of the message. The reception of messages is notified via a *recvMsg* notification containing the message identifier. Additional *opStart* and *opStop* notifications are sent every time an operation starts or stops processing a message, thereby marking the beginning and end of the atomic steps that compose an operation.

Since different operations may run on different threads, each thread is responsible for sending the notifications related to the operations it executes and to the messages that these operations produce. The *sendMsg*, *opStart* and *opStop* notifications must then be acknowledged by the debugger for the thread to continue executing its operations. By holding a specific acknowledgment, the debugger may therefore suspend the execution of the corresponding thread while the rest of the application keeps executing. The debugger also uses acknowledge messages to transmit information back to the application and to influence the future computation steps.

Suspending the execution of threads until the reception of an acknowledgement also guarantees that the debugger receives causally dependent notifications in the correct order. An operation does not send a message over the network before the debugger received and acknowledged the corresponding *sendMsg* notification. Messages sent to a given processing thread may be received by different communication threads, which send each a corresponding *recvMsg* notification after adding the message to the thread's pending message queue. Since we use of a single TCP connection between the debugger and each application instance, and since TCP guarantees that data sent over a single connection is not reordered, the order of reception of *recvMsg* notifications at the debugger matches the order in which messages are delivered to the thread. Similarly, the debugger cannot receive an *opStart* notification before the corresponding *recvMsg* notification.

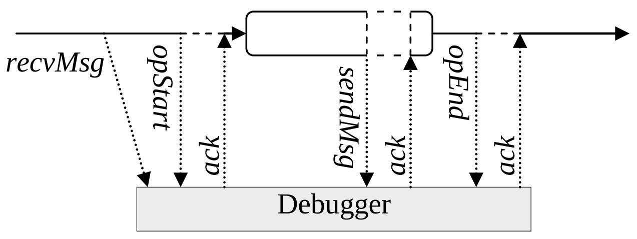
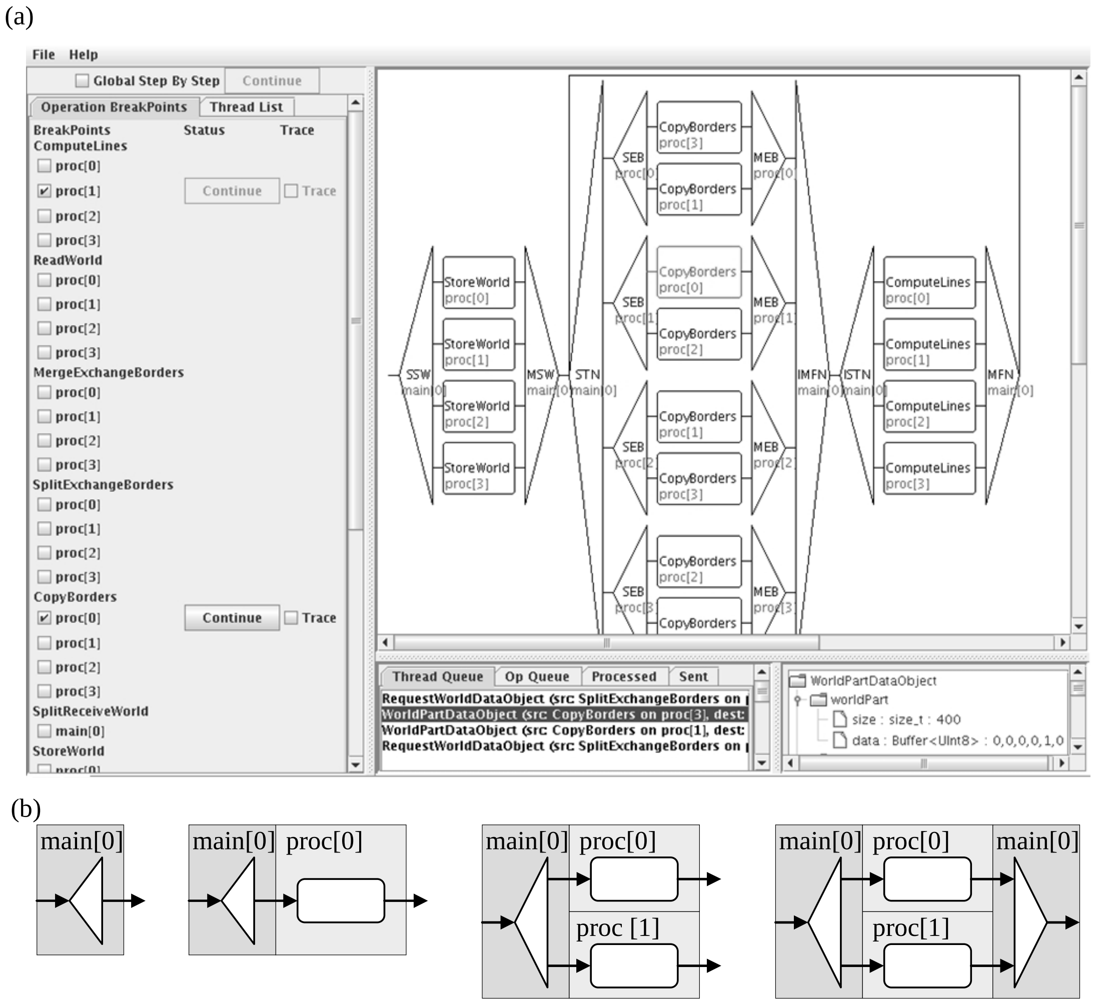


Figure 4.4: *Notifications sent by each operation to the debugger. The* sendMsg *notification includes the message being sent;* opStart *includes the identifiers of the operation and of the processed message;* opStop *includes the operation identifier;* recvMsg *includes the received message identifier.*

4.3.2 Features and functionality

The user interface of the debugger is shown at the top of Figure 4.5. The main area displays the current state of the application in the form of its unfolded flow graph. The lower part of the figure illustrates one possible execution scenario for a simple split-leaf-merge flow graph. Operation names and the thread they run on also appear to identify the different operations. The view is updated every time the debugger receives a notification. Operations are drawn in different colors to indicate their status, such as *idle*, *breakpointed* or *running*.

When the application starts, the debugger holds the acknowledgment for the input message of the first operation in the flow graph. A *Continue* button then enables the developer to resume the application execution. From that point, the default behavior of the debugger is to immediately acknowledge all notifications, allowing the application to continue until completion. Several mechanisms are provided to control the execution. The first is the *Global Step-by-Step* mode. When enabled, the debugger holds all acknowledgments, thereby suspending all threads of the application. Pushing the *Continue* button then sends one acknowledgment to each thread,



BreakPoints	Status	Trace
ComputeLines		
☐ proc[0]		
☑ proc[1]	Continue	☐ Trace
☐ proc[2]		
☐ proc[3]		
ReadWorld		
☐ proc[0]		
☐ proc[1]		
☐ proc[2]		
☐ proc[3]		
MergeExchangeBorders		
☐ proc[0]		
☐ proc[1]		
☐ proc[2]		
☐ proc[3]		
SplitExchangeBorders		
☐ proc[0]		
☐ proc[1]		
☐ proc[2]		
☐ proc[3]		
CopyBorders		
☑ proc[0]	Continue	☐ Trace
☐ proc[1]		
☐ proc[2]		
☐ proc[3]		
SplitReceiveWorld		
☐ main[0]		
StoreWorld		

```
RequestWorldDataObject (src SplitExchangeBorders on
WorldPartDataObject (src CopyBorders on proc[3], dest:
WorldPartDataObject (src CopyBorders on proc[1], dest:
RequestWorldDataObject (src SplitExchangeBorders on
```

```
WorldPartDataObject
  worldPart
    size : size_t : 400
    data : Buffer<UInt8> : 0,0,0,0,1,0
```

Figure 4.5: *(a) The graphical user interface of the debugger displays the current state of the application in the form of its unfolded flow graph; (b) the graph is drawn dynamically as the debugger receives notifications from the application.*

which then executes until it sends another notification to the debugger. Steps are therefore taken at the DPS operation level rather than at the code instruction level. This mode allows advancing quickly through the execution while still allowing the developer to take action on every notification.

The second execution control mechanism offers a finer grain of control using *operation breakpoints* that break on *opStart* notifications from a particular operation running on a particular thread. The debugger derives these breakpoints from the flow graph and the thread collections sent by the application: given an operation in the application flow graph and its thread collection, the debugger displays the list of threads within which the operation may run. Each operation breakpoint has an associated box, which may be checked to set the breakpoint and instruct the debugger to hold the acknowledgments of the matching *opStart* notifications. The *Continue* button next to each activated operation breakpoint is enabled when the breakpoint is hit, i.e. when the debugger receives a matching *opStart* notification. Pressing the button then resumes the operation execution. The left part of Figure 4.5a shows the list of operation breakpoints for the flow graph of our neighborhood-dependent computation example.

While looking for a bug, it is often useful to study one particular path in the application. That is, when looking at an operation we follow its output message(s) to the successor operation, then to the next and so on, while ignoring what is happening in the rest of the application. Such a behavior is enabled by *tracing* the messages generated by a breakpointed operation. This is done by checking the right-hand side box of an operation breakpoint. When the debugger resumes the execution, the *tracing* flag is piggybacked on the acknowledgment, causing all the *sendMsg* notifications generated by the operation to have a tracing flag set. From this moment onwards, when the debugger receives an *opStart* notification containing the identifier of a traced message, it automatically sets the breakpoint of the triggered operation and holds the acknowledgment. All the successors of an operation in the unfolded flow graph are therefore automatically breakpointed, enabling the developer to navigate through one particular branch of the graph. The trace box can then be unchecked individually for every operation breakpoint, allowing the programmer to focus on the problem he is debugging. This is particularly helpful when a traced message enters a split operation. Since all the outputs of the split will be traced, many operation breakpoints will be set simultaneously.

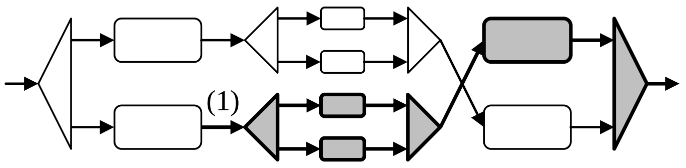
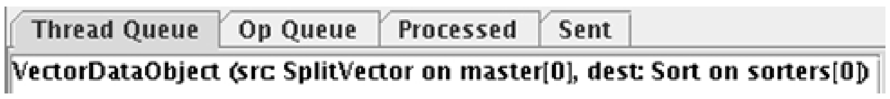


Figure 4.6: *Tracing a message automatically breakpoints all its successors, thereby enabling the developer to focus on a single flow graph branch. In this example, tracing is enabled on message (1).*

4.3.3 Influencing the application execution

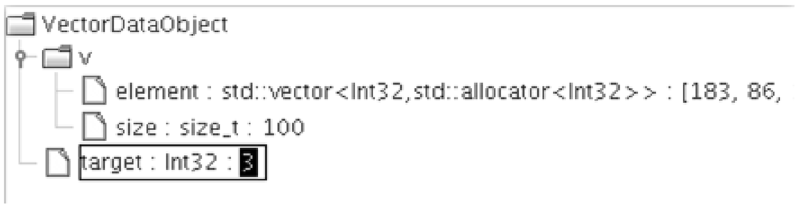
DPS threads queue incoming messages until they may be processed (Section 2.5). The developer can explore the queues of the various threads (Figure 4.7a, *Thread Queue*), and reorder the messages they contain. This changes their processing order by the DPS thread and allows testing the application for message races. Modified orderings are transmitted back to the thread along with the acknowledgment, and messages are accordingly reordered within the thread queue. That feature can be used by breakpointing an operation on a single thread: as the execution of other threads is unhindered, all the messages they send to the thread running the breakpointed operation will accumulate in its pending message queue, providing the developer with an instantaneous view of a set of messages that may race.

(a)



```
VectorDataObject (src SplitVector on master[0], dest: Sort on sorters[0])
```

(b)



```
VectorDataObject
  v
    element : std::vector<Int32,std::allocator<Int32>> : [183, 86,
    size : size_t : 100
  target : Int32 : 3
```

Figure 4.7: *(a) Message lists display the type of the message, as well as the source and destination operation, and (b) selecting a message displays the message content. The value of target is being edited. Here, the vector is an actual C++ standard library vector.*

At any moment, the developer may select an operation in the flow graph. This updates the content of the three remaining tabs in Figure 4.7a. The *Op Queue* tab displays the messages from the thread queue that are to be processed by the selected operation. The *Processed* tab lists all the messages which have been received and processed by this operation since the beginning of the application execution. Finally, the *Sent* tab displays all the messages which have been sent by the operation. The queues identify each message by displaying its type, as well as the name of the source and destination operations and threads.

Selecting a message in any one of the lists displays its content in a tree view similar to the ones found in traditional sequential debuggers (Figure 4.7b). This is made possible by using the specialized textual serializer described in Section 2.7.4 to transfer messages from the application to the debugger. The textual serialization avoids byte-ordering and internal data representation issues between hosts running the C++ DPS application and the Java debugger. Therefore, the debugger and the parallel application can run on different operating systems and hardware. Since the debugger has no knowledge about the types and structure of the data objects used by the application, the textual serializer adds the necessary typing and variable name information to the data.

If the developer selects an operation that is suspended on a *sendMsg* notification, the corresponding message is highlighted in the Sent list. The *sendMsg* notifications are sent before delivering the message to the communication layer. The developer may therefore modify the message from within the debugger before its transfer to the next operation. The modified message is then sent back to the suspended thread together with the acknowledgment. The thread then discards the original message and replaces it with the one received from the debugger. This scheme allows the developer to alter messages so as to modify the behavior of the application. One example is to modify fields used by the routing function to determine the destination thread of a message; the developer could change their value to test whether his application makes any assumption about which thread will process the message. A similar mechanism can be used to display the content of local thread storage objects within the debugger.

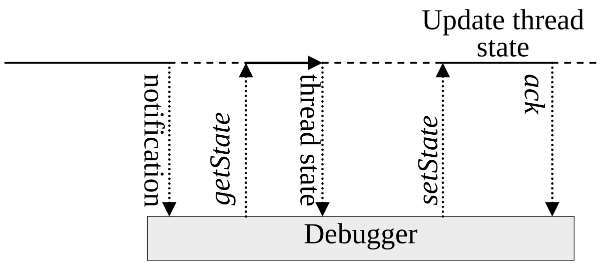


Figure 4.8: *Threads suspended while waiting for debugger acknowledgments can receive requests, e.g. to modify the ordering of pending messages or update the content of thread state objects.*

4.3.4 Debugging example

This section shows how the debugger can be used to discover a message race within the neighborhood-dependent parallel application presented in Section 4.2. The goal is to use the debugger to produce the execution depicted in Figure 4.3.

In order to test the application, the developer sets an operation breakpoint on the split operation on the thread $P[1]$. When $P[1]$ is about to start the split operation that sends *Send border* requests, it sends an *opStart* notification to the debugger, which hits the breakpoint. As the other threads keep executing, their messages requesting the borders appear in the pending message queue of $P[1]$. The developer then moves the two requests in front of the input message of the split operation in the queue. Once the new ordering is uploaded to the thread and the messages have been reordered accordingly in the queue, $P[1]$ sends back a new *opStart* notification saying it is about to start executing a *Send border* operation. Since this operation is not breakpointed, and assuming that the *Global Step-By-Step* mechanism is not enabled, the notification is immediately and automatically acknowledged. The execution of $P[1]$ is suspended once again when both border exchange requests have been processed. At that point, both $P[0]$

and *P*[2] have received the borders from their neighbors and start updating their part of the domain.

From that moment, no matter when *P*[1] is resumed, its neighbors will process its requests for borders after they have updated their state. Determining whether the race exists can be done either by looking at the final state of the thread, or by looking at the content of the messages containing the borders received by *P*[1].

4.3.5 Scalability issues

Since it must receive, process and acknowledge all the notifications sent by the application threads, the debugger may quickly become a bottleneck when the number of threads grows. Measurements for the neighborhood-dependent computation running on 2 compute nodes show that the parallel application runs 170 times slower when the debugger is active and iterations are short (i.e. 22 milliseconds per iteration). The slowdown drops to 2.7 with more intensive computations (i.e. 2 seconds per iteration). The corresponding slowdown factors are respectively 390 and 9 when the application runs on 8 compute nodes. Since the overhead is directly proportional to the number of notifications sent to the debugger, it is particularly important for applications performing many short-lived operations.

In the current implementation, the debugger must receive every notification in order to evaluate message and operation breakpoints. By distributing parts of the debugger functionality (such as breakpoint evaluation) within the threads, or within debugger servers running on the compute nodes [49, 74, 114], it would no longer be necessary to systematically send all messages and all notifications to the debugger. If the debugger does not need to continuously display updates to the application state, many notifications can be sent at once and then only induce updates to the displayed graph and the message queues. The full functionality would then only be enabled on demand for specific application parts.

Another challenge consists in displaying the partial unfolded flow graph information for a large number of threads. Matching split-merge pairs of operations could be collapsed into a single node to reduce the size of displayed graph.

4.4 Automated testing through static decomposition

Manual testing provides much information and full control to the developer. However, it is error prone and time consuming. Moreover, the overwhelming number of possible executions makes it impossible to test them all. Developers are therefore condemned to focus their effort on a subset of cases, with the risk of overlooking potential errors. In this section, we focus on the automated testing of DPS applications, using a static graph decomposition technique for reducing the computational complexity of the testing process.

This section only considers parallel applications producing a *fixed message set*, i.e. applications that generate the exact same set of messages independently of the ordering in which these messages are delivered[2]. Messages must have the same content and the same source and destination in all runs. Most linear algebra computations and finite elements methods have that property. On the contrary, applications such as divide-and-conquer optimization problems where the current best solution influences the remainder of the computation, as well as applications that dynamically route messages to different threads to balance the load do not belong to that category.

We recall our underlying assumptions that the only non-determinism in the execution of an application lies in the ordering in which messages are delivered, and that DPS operations do not modify memory accessible from other DPS threads. Under these assumptions, each parallel execution has at least one equivalent serial execution, defined by a specific ordering of message delivery. We therefore want to test that all message orderings and their associated serialized computations yield the same results.

4.4.1 Building a Partial Order Execution Graph

Given the execution trace of a DPS application, we can easily determine the causal relationships between the messages sent during that particular execution. Like in the previous chapter, we decompose all operations into *atomic steps* to represent parts of operations that are executed atomically (Figure 4.9a). The difference is that we assume that split and stream operations can no longer be suspended upon sending a message, i.e. that flow control (Section 2.6.2) is disabled. Both the leaf and split operations therefore consist of a single atomic step, while merge and stream operations are decomposed into one atomic step per input message. This decomposition ensures that the number of messages is the same as the number of atomic steps, and that admissible message orderings are equivalent to admissible orderings of atomic step executions (Figure 4.9b). The processed unfolded flow graph is called the Partial Order Execution Graph (POEG) of the application execution [20], where edges represent Lamport's *happened-before* relationship [64]. In our context, a message a is delivered before b if a is a predecessor of b in

[2]We will relax this assumption in Section 4.5.

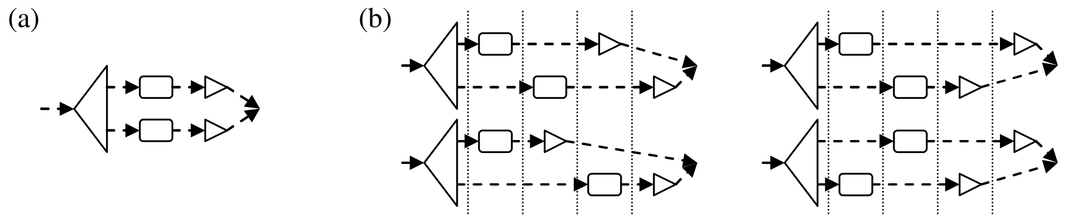


Figure 4.9: *(a) Partial Order Execution Graph for a split-leaf-merge flow graph where the split sends two messages, and (b) four serialized executions associated to different admissible message orderings.*

the POEG.

For merge operations, all atomic steps are predecessors of the operation triggered by the sole output message. For stream operations however, the dependencies between the multiple outputs and inputs are not readily available. Assuming that these dependencies are fixed, we determine them by reexecuting a stream operation multiple times for different orderings of inputs. When all messages but one have been delivered, all output messages that do not depend on the missing input message have already been sent. Delivering the last message of a stream operation thus triggers all the outputs that depend on that message. We repeat this scheme once for every input message and combine the collected dependencies to produce the required information. All dependencies of a stream taking n inputs can therefore be fully determined using only n permutations of inputs (Figure 4.10). Since we distinguish messages based on their identifier, it is particularly important that message identifiers produced by stream operations are also independent of the ordering of inputs (Section 2.5.1).

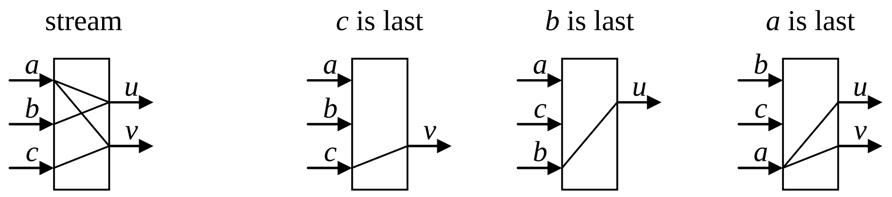
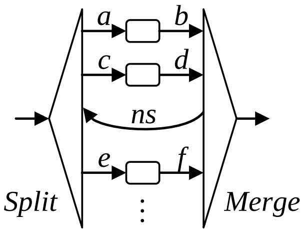


Figure 4.10: *The dependencies of the stream operation on the left (u requires a and b, v requires a and c) can be derived by looking at which messages are output by the operation while delivering the last input message.*

The reason for disabling the flow control functionality of DPS within the application is the following. A flow controlled split operation executes in multiple parts, each of which produces a set of messages. Each part is triggered by the reception of a *NotifySplit* from the merge operation, which in turn depends on the messages received by the merge. This sequence induces causal dependencies that are dependent on the ordering of the delivery of messages to the merge operation (Figure 4.11). If the flow control is disabled, the split operation executes without being suspended, and it can output all its messages at once. The message dependencies within the flow graph are therefore statically defined and can be captured within a POEG.



$a \rightarrow b \rightarrow ns \rightarrow e \rightarrow \ldots$
$c \rightarrow d \rightarrow \ldots$

If the merge first receives *b*, *e* is a successor of *b*.

$c \rightarrow d \rightarrow ns \rightarrow e \rightarrow \ldots$
$a \rightarrow b \rightarrow \ldots$

If the merge first receives *d*, *e* is a successor of *d*.

Figure 4.11: *When flow control is active, the dependencies between messages depend on the ordering of their delivery in the merge or stream matching the flow controlled operation.*

Building the POEG of a DPS application thus requires three assumptions: (1) the application produces a fixed set of messages, (2) the message dependencies induced by stream operations are independent of the ordering of input messages, and (3) that flow control may be disabled without changing the fundamental behaviour of the application. The impact of the fixed message set assumption is discussed in Section 4.4.2. The impacts of the second and third assumption, as well as the consequences of violating them is discussed in Section 4.4.7.

4.4.2 Partitioning the POEG

In the general case, the number of admissible orderings grows exponentially with the number of messages sent, making it impossible to test them all. Many orderings can however be prevented by partitioning the POEG into subgraphs representing parts of the application execution, such that each part can be processed independently from the others.

Swapping the delivery order of two messages that are delivered to different DPS threads will not change the computation outcome. Our first step is therefore to separate the contributions from atomic steps running on different threads. The ability to test each thread independently relies on our assumption that the application produces a fixed set of messages. Indeed, this assumption guarantees that the set of input messages of each thread is unique. Since messages travel from one thread to another, if one thread can produce different sets of output messages, the other threads have to be tested for multiple sets of inputs and can no longer be studied independently from each other.

If every thread produces a fixed set of output messages for all orderings of its input messages, then the application contains no message race. In contrary, finding an ordering of the inputs causing a thread to produce a different set of output messages implies that the application being tested does not produce a fixed set of messages. Determining whether the produced outcome is erroneous or not is left to the developer. If it is erroneous, we have found a message race. If it is not, the application being tested does not produce a fixed message set, and the contribution of the different threads may not be tested independently from each other. The testing method therefore cannot declare an application to be race free if the fixed message set assumption is incorrect.

Given the POEG of a complete execution, we obtain the POEG of each thread by removing all the messages delivered to other processes, while maintaining the causality between messages delivered to the thread under consideration. The POEG of each thread therefore defines the admissible orderings of all computations performed in the same memory space. Figure 4.12 illustrates this principle by isolating the contribution of thread *P*[1] from Figure 4.2. When focusing on a single thread, it becomes clear that the execution order of the *Send border* and *Update* operations is not constrained, and that a message race might occur.

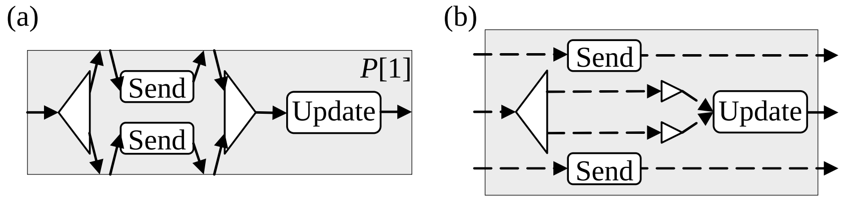
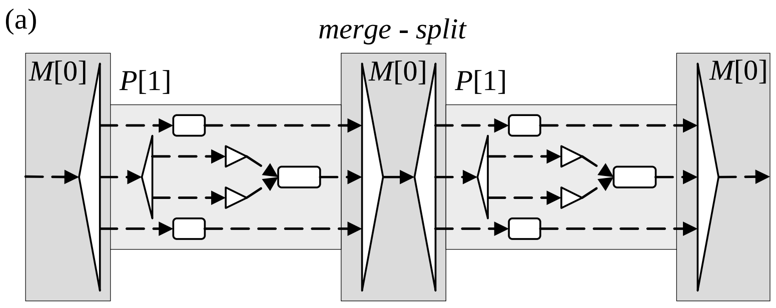


Figure 4.12: *(a) Operations running on thread* P*[1] of Figure 4.2, and (b) the POEG of* P*[1], after decomposition into atomic steps and removal of messages delivered to other processes.*

Within a thread, the causality between messages can prevent distinct subgroups of messages from being interleaved. Figure 4.13 illustrates this principle on two iterations of our neighborhood-dependent computation. The synchronization enforced by the merge-split construct found between the two iterations is represented within the POEG by the fully interconnected dependencies between the atomic steps preceding the merge and the ones following the split operation (Figure 4.13b).

In order to identify these subgroups in the POEG of each thread, we first identify sets of atomic steps that are fully interconnected. Two sets of atomic steps S_1 and S_2 are fully interconnected if every atomic step $s_i \in S_1$ is a predecessor of every atomic step $s_j \in S_2$. We then introduce an auxiliary atomic step between every such pair of sets (dark gray node in Figure 4.14a), and add a source and sink node to the graph (s and t). We run a unit flow through the graph, such that the output flow of each atomic step is split equally between each successor, and the contributions of multiple input flows to a single atomic step are added. When the sum of the input flows into an atomic step is one, the number of messages delivered before and respectively after its execution is constant. We are therefore allowed to split the messages sent



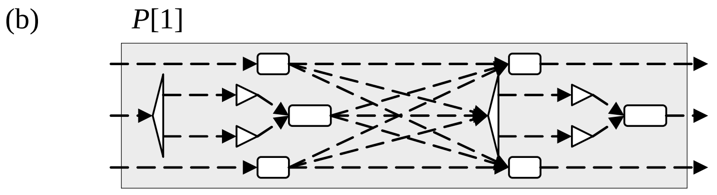
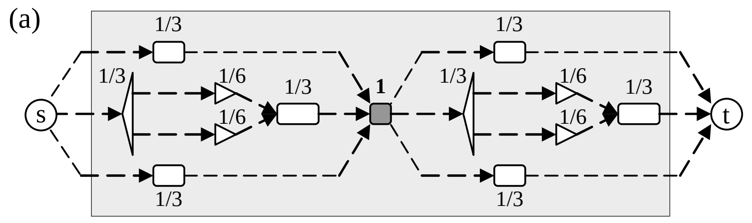


Figure 4.13: *(a) POEG of two iterations of the neighborhood-dependent computation illustrated in Figure 4.2 (*P*[0],* P*[2] not shown), and (b) equivalent POEG of* P*[1].*



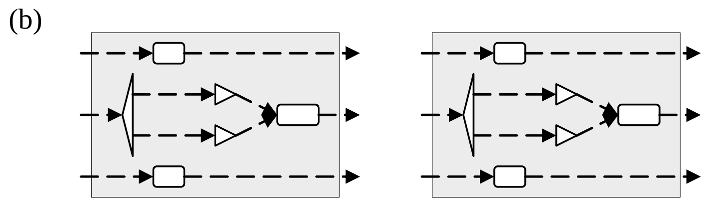


Figure 4.14: *(a) Introduction of an auxiliary atomic step (dark gray), and of source and sink nodes* s *and* t*, and (b) partition of the POEG into subgroups.*

before and after such an atomic step into two consecutive subgroups, each one with its own POEG (Figure 4.14b).

4.4.3 Partial-Order Reductions

Within a partial order execution graph, orderings can be prevented by inserting additional edges to force the relative delivery order of specific messages, or equivalently, the relative execution order of the atomic steps they trigger.

One reason for preventing orderings is that some of the ones allowed by the POEG may never occur in practice. Failure to take this into account may lead to false positives, where spurious errors are detected. For instance, message-passing libraries may guarantee a FIFO delivery of messages. Applications that rely on that assumption may send subsequent pieces of data without explicit synchronization between successive messages. In order to account for that assumption within the POEG, we may, when the delivery of a single message causes multiple messages to be sent, add ordering constraints between the messages that are to be delivered to the same destination thread.

Within subgroups, distinct message orderings may also be equivalent. Let us assume that we know for each operation (and by extension for each atomic step and the messages that trigger them) if the state of the thread is only read or if it is modified. If two atomic steps only read the unmodified state, the order of their execution has no impact on the final thread state. If two messages trigger such atomic steps, and if the POEG defines no causality between them, we say that the messages are *exchangeable*. However, we cannot constrain their delivery order without taking their successors and predecessors into account. Indeed, if we constrain a to be delivered before b, we transitively constrain all predecessors of a (noted $Pred_a$) to be delivered before all successors of b (noted $Succ_b$). An edge $a \rightarrow b$ can therefore be added to the POEG only if every message in $Pred_a \backslash Pred_b \cup \{a\}$ is exchangeable with all messages in $Succ_b \backslash Succ_a \cup \{b\}$, where $\backslash$ denotes the set difference operation. The partial-order reduction of a POEG is therefore performed by adding all the edges that satisfy this condition.

Computations that modify different variables in the thread state may also be commutative. Detailed memory access information can be represented using *access vectors*, which specify whether each DPS operation reads, writes or ignores each variable of the thread state. Each atomic step, and by extension the message that triggers it, inherits from the access vector of its associated operation. The partial-order reduction described above can then be applied, using a generalized definition for the exchange of messages: two messages can be exchanged if all the members of their access vector can be exchanged (Figure 4.15).

Variable Access Exchange rules

$$\begin{matrix} a \\ b \\ c \end{matrix} \quad \underset{v_1}{\begin{bmatrix} \mathrm{r} \\ \mathrm{w} \\ - \end{bmatrix}} \; \underset{v_2}{\begin{bmatrix} \mathrm{r} \\ - \\ - \end{bmatrix}} \; \underset{v_3}{\begin{bmatrix} \mathrm{r} \\ \mathrm{r} \\ \mathrm{w} \end{bmatrix}} \qquad \begin{matrix} (\mathrm{r}) \longleftrightarrow (-) \\ (\mathrm{r}) \longleftrightarrow (\mathrm{r}) \\ (\mathrm{w}) \longleftrightarrow (-) \end{matrix} \qquad \begin{matrix} \mathrm{r}: \text{read} \\ \mathrm{w}: \text{write} \\ -: \text{no access} \end{matrix}$$

Figure 4.15: *Three possible access vectors for a thread with three member variables a, b and c in its local storage. Member variables can be read (r), modified (w), or ignored (-). According to the exchange rules on the right, the message associated to v_2 can be exchanged with those associated to v_1 and v_3.*

An application of the algorithm is illustrated in Figure 4.16a, where edges are added between the messages triggering the read-only atomic steps (1), (2) and (3). The orientation of the new edges is important. An edge (2)→(1) would not satisfy our constrains: it would prevent orderings where (2) is executed after the non-read-only *Update* operation, which are the exact cases that we want to test.

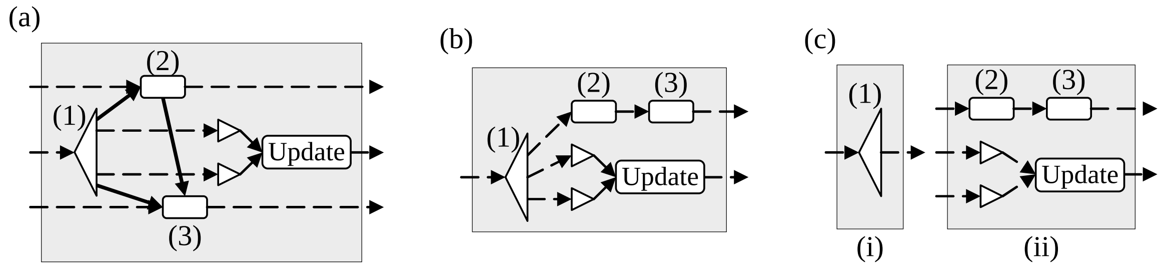


Figure 4.16: *(a) POEG of thread* P*[1] after partial-order reduction, (b) transitive reduction and (c) subgroup partitioning.*

DPS stream and merge operations, which are composed of multiple atomic steps, may maintain their own internal state in addition to the underlying thread state[3]. It is therefore important that we test the possible orderings of their input messages. Since the access vector only stores operation accesses to the local thread storage, it does not provide the full information about the commutativity of atomic steps belonging to such operations. The partial-order reduction therefore avoids adding edges that introduce ordering constraints between messages processed by the same operation.

The adjunction of edges in the POEG may cause some pairs of messages to be connected through several paths. Performing the transitive reduction of the POEG removes the superfluous edges (Figure 4.16b), and may enable a finer-grain subgroup partitioning. In Figure 4.16c, the message triggering the atomic step of the split operation now always occupies the first position and can therefore be used to further partition the subgroup. With this final partition, the number of admissible message orderings is 1 for the subgroup (i), and 20 for the subgroup (ii), versus 168 orderings for the POEG of Figure 4.12b.

Each subgroup is tested for message races by executing multiple different orderings, and by checking that the final process state and the set of generated messages are identical in all cases. The developer may decide for each subgroup whether to test all or only a subset of possible orderings.

[3]See for instance the *MergeVector* operation in Section 2.4.2.

4.4.4 Generating a subset of possible orderings

Applications with few synchronizations and few read-only operations may benefit little from the decomposition into subgroups. Moreover, it is not always practical to test every single ordering when computations have a significant duration. Fortunately, a single bug generally causes races in many different orderings. Within the subgroup of Figure 4.12b for instance, the error is revealed as soon as one of the two borders is sent after the subdomain update, which occurs in 24 orderings out of the 168 admissible orderings, or 14% of the orderings. After the decomposition into subgroups, this percentage grows to 40%, or 8 out of the 20 admissible orderings (subgroup (ii) in Figure 4.12c). We therefore suspect that most message races can be revealed by testing a small subset of carefully selected orderings, generated as described below.

Since we want to ensure that the final state of the local thread storage is identical for all orderings, and since the final state is determined by the last atomic step that modified it, we generate one ordering for each message such that the message is delivered as late as possible. Formally, given a subgroup of messages S we generate the following orderings:

$$\{(S \backslash \{a, \mathit{Succ}_a\}, a, \mathit{Succ}_a) \mid a \in S\} \tag{4.1}$$

Parentheses denote an ordered list. Expression 4.1 means that, for every message a in S, we generate an ordering where all messages other than the successors of a in the POEG of S are delivered before a. We may thus test whether delaying as much as possible the execution of each atomic step has an impact on the final state of the thread. This is similar to the *halt-one-thread* heuristic in ConTest [30].

Our second goal is to test that within a subgroup all orderings produce a fixed set of messages. We therefore generate orderings such that every message is delivered right after every other message, and both are delivered as early as possible in order to test their influence on the following computations. Formally, if Pred_a is the set of predecessors of a in the POEG of S, and $\mathit{Pred}_{a,b}$ is $\mathit{Pred}_a \cup \mathit{Pred}_b$ we generate the following set of orderings:

$$\{(\mathit{Pred}_{a,b}, a, b, S \backslash \{a, b, \mathit{Pred}_{a,b}\}) \mid a, b \in S, a \neq b\} \tag{4.2}$$

If the delivery order of two messages a and b is not constrained by the flow graph, expression 4.2 specifies at least one ordering containing $(\ldots, a, b, \ldots)$ and one ordering containing $(\ldots, b, a, \ldots)$. The rationale here is to check that the output messages sent by an atomic step are the same no matter which other atomic step (which could modify the thread state) was executed right before.

If $|S|$ is the number of messages in the subgroup S, Expression 4.1 generates $|S|$ orderings. However, since we only consider orderings that are admissible according to the POEG of the subgroup, Expression 4.2 produces less than $|S| \cdot (|S| - 1)$ distinct orderings in the general case.

4.4.5 Implementation

We implemented the Partial Order Execution Graph decomposition and testing mechanism on top of the simulator described in Chapter 3, which already provides most of the infrastructure needed to fully control the execution of an application. Tested applications therefore run within a single process, where all computation threads run under the control of the simulator thread. A *validator* object is responsible for handling the whole testing process.

The decomposition and the analysis of the POEG is based on an execution trace. We run the application once and log the messages sent within the validator. In order to be able to replay specific orderings, we keep a full copy of every message. The additional information stored within the messages (Section 2.5.1) is sufficient to determine which operations were triggered, as well as the causal dependencies between the messages. We also use the checkpointing capabilities of DPS to keep a copy of the initial content of the local storage of each thread, which must therefore be made serializable as described in Section 2.7.2. The testing procedure starts before returning the final message to the caller of the flow graph, ensuring that all threads remain active and are able to execute further operations.

In order to enable the partial order reduction presented in Section 4.4.3, we must collect additional information about how operations access the local thread storage. The vector indicating modified variables is obtained by comparing checkpoints of the thread state taken immediately before and after the execution (Section 2.7.5). However, collecting *read* information requires some help from the developer. All calls to the *getThread* function (Section 2.4.5) for accessing the local thread storage must be replaced by a macro, which transfers the name of the member variable being accessed to the validator during the initial run.The validator then builds a vector containing *read* information for the executed operation.

The serializable object comparison mechanism also provides the names of all the object members. The collected *read* information may therefore be added to the thread state comparison vector. The resulting access vector is associated with the name of the C++ operation class. If there are different execution paths within an operation, the access vector may be different in multiple executions of that operation. The information collected from running the same operation multiple times is therefore combined conservatively, e.g. a variable read in one operation instance and written in another is marked as written. All the messages delivered to operations of the same type therefore share the same access vector.

When the initial run completes, we first check that the outputs of split and stream operations have distinct identifiers (Section 2.5.1), which guarantees that all identifiers are indeed unique within the execution trace. We then decompose merge and stream operations into multiple atomic steps, and add edges in the POEG to represent the dependencies between them.

If the merge and stream operations run on a thread without local storage, the validator also checks that permutations of inputs always produce the same set of output messages. Executing one permutation simply requires delivering the messages to the appropriate DPS thread. The thread creates the operation upon receiving the first input message. As in a regular execution, the validator delivers a *NotifySplit* message (Section 2.6.1) to the operation. All such notifications are collected during the initial run and delivered to the operations during testing. The operation therefore knows how many messages to expect and eventually completes after the reception of the last message.

Once the decomposition has been performed, the validator builds the POEG containing all the executed atomic steps. It then applies the partial-order reduction and partitions the POEG as described in the previous sections. Since the FIFO link property is not always guaranteed by the DPS runtime, we do not consider this optimization in our study.

When the orderings of the first subgroup of a thread are being tested, the initial thread state is recovered from a checkpoint before the execution of each ordering. Another checkpoint is taken once all the messages have been delivered, and serves as a reference for verifying that all orderings lead to the same final thread state. It also provides the initial thread state for testing the next subgroup. Generated messages are matched against the reference execution using the message identifier. New messages are compared to the reference messages as soon as they are produced.

Testing subgroups requires comparing the sets of output messages and the final state of local thread storage objects. Since all comparisons are performed using the serializable object comparator described in Section 2.7.5, the developer may provide custom comparison functions if the default binary comparison is unsuitable. In practice, the developer must also ensure that all fields of all serializable objects are initialized. Uninitialized fields otherwise take random values that make all objects different. Such cases are however quickly identified by looking at the actual content of values that are labeled as different by the serializable object comparator.

When an ordering leads to a different final thread state or message set, the reference input and output messages, the initial process state and the ordering are written to stable storage together with the reference ordering. The stored information can then be used to replay the ordering that caused the race. Since executions are replayed within a single multithreaded process, a conventional debugger can be used to study the erroneous computations.

4.4.6 Results

Let us present practical results for a few parallel applications. In order to take into account both the number of orderings as well as their length, the metric used for all measurements is the total number of messages that must be delivered, or equivalently of atomic steps that must be executed, in order to test all possible orderings. This number is obtained by summing for all subgroups the number of orderings allowed by their POEG multiplied by the number of messages within the subgroup.

We first quantify the benefits of the partial order reduction and subgroup decomposition using the neighborhood-exchange application described in Section 4.2. Table 4.1 compares the number of delivered messages for exhaustively testing two iterations of the neighborhood-exchange computation, when no decomposition is performed, when using only the thread decomposition (TD, Section 4.4.2), when adding the partial order reduction (TD+POR, Sections 4.4.2 and 4.4.3), and when performing the full subgroup decomposition (Sections 4.4.2 and 4.4.3). We see that it is impossible to execute all orderings without decomposing the POEG of the application, even when it runs on only two threads. The proposed optimizations reduce the number of atomic steps to be executed by a factor of 10^{13}. In practice, testing all orderings for an application run of 6 iterations on 8 threads takes about 8 seconds on a 2.4GHz Pentium 4 processor.

We carry out the same analysis for a parallel implementation of the Floyd-Steinberg halfton-

Table 4.1: *Total number of messages to be delivered in order to test all orderings (neighborhood-exchange application with two iterations).*

	2 threads	4 threads	6 threads
No decomposition	$5.6 \cdot 10^{16}$	–	–
Thread decomposition (TD)	$2.4 \cdot 10^{5}$	$4.9 \cdot 10^{5}$	$10 \cdot 10^{6}$
TD + partial order reduction (POR)	$2.9 \cdot 10^{4}$	$6.6 \cdot 10^{4}$	$9.4 \cdot 10^{6}$
Subgroup decomposition + POR	860	1932	$1.3 \cdot 10^{4}$

Table 4.2: *Total number of messages to be delivered in order to test all orderings (parallel Floyd-Steinberg halftoning algorithm).*

	2 threads	4 threads	6 threads	8 threads
No decomposition	$6.8{\cdot}10^8$	–	–	–
Thread decomposition (TD)	848	$3.5{\cdot}10^5$	$6.5{\cdot}10^8$	$4.0{\cdot}10^{12}$
TD + partial order reduction (POR)	116	$1.9{\cdot}10^4$	$1.9{\cdot}10^7$	$7.5{\cdot}10^{10}$
Subgroup decomposition + POR	42	1280	$7.4{\cdot}10^4$	$6.8{\cdot}10^6$

Table 4.3: *Total number of messages to be delivered in order to test all orderings or the subset defined in Section 4.4.4 (pipelined parallel LU factorization).*

	3 threads	4 threads	5 threads
Subgroup decomposition + POR	$7.9{\cdot}10^5$	–	–
Subset defined in Section 4.4.4	2773	18362	82353

ing algorithm [8], which converts a grayscale image into a black and white image. It determines for each grayscale pixel whether it should be black or white. The error, i.e. the difference between the desired gray value and the selected binary value, is then added according to an error-diffusion weight matrix to the gray value of the unprocessed neighboring pixels. Table 4.2 summarizes the results. For 2 threads, the full decomposition reduces the number of atomic steps that must be executed by a factor of 10^7 compared to when no decomposition is performed. Testing all orderings for an application run on 8 threads takes about 105 minutes for a grayscale image of size 256×256 pixels.

Finally, let us examine a parallel block LU factorization application (Table 4.3). Since the iterations of the computation are loosely synchronized in order to maximize the pipelining of the computation, subgroups contain many messages with little dependencies between each other, causing the number of atomic steps to be executed to explode. It remains however possible to test a subset of possible orderings using the algorithm described in Section 4.4.4, as shown by the last line of Table 4.3. The partial test for a 160×160 matrix with 5 threads takes about 30 minutes.

In order to test our message race detection software, we artificially introduced races by removing code ensuring a correct processing of out-of-order messages within merge and stream operations. We also discovered a few genuine bugs in previous implementations of the LU factorization application. We compared the results of the partial and full testing in all cases where the latter could be performed. In the applications presented here, testing the orderings produced by the algorithm described in Section 4.4.4 was sufficient to find every message race.

4.4.7 Limitations

Using a single multithreaded process to test all orderings limits the size of instances that may be tested. In our experiments, storing the whole trace of the parallel Floyd-Steinberg application only requires 4.4MB when processing a 256×256 image on 8 threads, but looking for races in the LU factorization application of a 160×160 matrix on 10 threads requires 350MB of memory.

Since different threads and different subgroups are tested independently from each other, we could test them in parallel, thereby further reducing running times and memory consumption. One possibility would be for each thread to log the messages it sends and receives during the reference execution. A master would collect partial traces from every thread, build the complete POEG, perform the decomposition and send the subgroups and their associated POEGs back to the computation threads. All threads would then have enough information to test their own subgroups in parallel.

In the beginning of this section, we described three assumptions enabling us to build the POEG of a DPS application. The first assumption states that the application must produce a fixed message set. Violating the assumption causes the testing mechanism to detect orderings producing different messages sets. It is therefore the responsibility of the developer to determine whether reported errors are actual message races or are simply a proof that an application does not produce a fixed set of messages.

The second assumption is that disabling the flow control mechanism does not change the behavior of the application. Since flow control introduces dependencies between the messages sent by a split operation, an application could rely on it to sequence specific messages. Disabling the flow control therefore causes our testing mechanism to test additional message orderings that may produce false positives. Similarly, our implementation of the traveling salesman solver (Section 3.6) uses flow control to enable the split operation to use better solutions received by the merge. Disabling the flow control mechanism changes the application behavior by causing the split to send a fixed set of messages in all cases. The POEG decomposition would therefore test a set of executions that only partially overlaps with those allowed by the original application.

Finally, we assume that the message dependencies caused by stream operations are statically defined, and are independent of the values stored within the thread local storage. Indeed, since the POEG is a static structure, it cannot represent such variable information. However, the main use of stream operations is to add synchronizations within the execution, and the vast majority of such synchronizations are strictly defined for a particular application. Nevertheless, there might be some cases where more flexible behaviors are appropriate. Such cases could be detected using the following extension: instead of only considering the messages output upon the delivery of the last input message, the validator could consider those produced for all inputs. For each input message, the validator can check that the set of messages output is consistent with the already available dependency information. Whether this scheme is equivalent to testing all permutations of inputs remains an open question however.

4.5 Dynamic testing

The major drawback of the static POEG decomposition is that it can only be applied to applications that produce a fixed set of messages, and whose parallel behavior does not rely on flow control or on variable message dependencies within stream operations. This section therefore presents another complementary method that does not suffer from such restrictions.

Like in the previous section, we assume that the only non-determinism in the execution of an application lies in the ordering in which messages are delivered, and that operations do not write to memory readable by other threads. Each parallel execution therefore has at least one equivalent serial execution, defined by a specific ordering of message delivery. We represent an execution as a sequence of states, where the transition from a state to the next is triggered by the delivery of a message. The transition ends upon completion of all the computations triggered by the delivered message. In our context, the state of the application is defined by the set of messages that have been sent but not yet received (i.e. the messages in transit), by the content of the local storage of every thread participating in the computation, and by the state of every suspended operation.

We may combine sequences corresponding to different orderings into a *message-passing state graph* by merging states common to different executions. Combining all possible sequences produces the *full* message-passing state graph of an application. Each path within the graph defines a different ordering of messages. A single state has multiple outgoing edges when more than one message is in transit, and has multiple incoming edges when it can be reached via several message orderings. Since in our execution model all computations are triggered by the delivery of a message, reaching a non-final state with no message in transit reveals a deadlock[4].

By exploring reachable states by actually executing the application code, this scheme fully supports the use of flow control and of load-balancing within the application (Section 2.6), and makes no assumption on which dependencies are induced within stream operations.

Figure 4.17 displays a simple example. Given the unfolded flow graph shown in (a), delivering the initial message triggers the execution of the split operation, which sends messages 2 and 3 during its execution. These two messages are therefore in transit when the operation terminates and the next state is reached. We may then deliver either message 2 or message 3. If the two leaf operations triggered by messages 2 and 3 execute on different DPS threads, they have access to variables stored within distinct memory spaces and cannot interfere with each other. Delivering message 2 before message 3 or message 3 before message 2 therefore leads to the same state with messages 4 and 5 in transit. Note that this example is a simplification of an actual execution, as it does not represent *NotifySplit* and *NotifyMerge* messages exchanged by the split and the merge operations; the *NotifySplit* is produced upon delivery of the first input of the merge operation, and the *NotifyMerge* is produced upon delivery of the *NotifySplit* notification. Considering these notifications like regular messages is necessary to correctly represent executions of applications relying on flow control or on load-balancing.

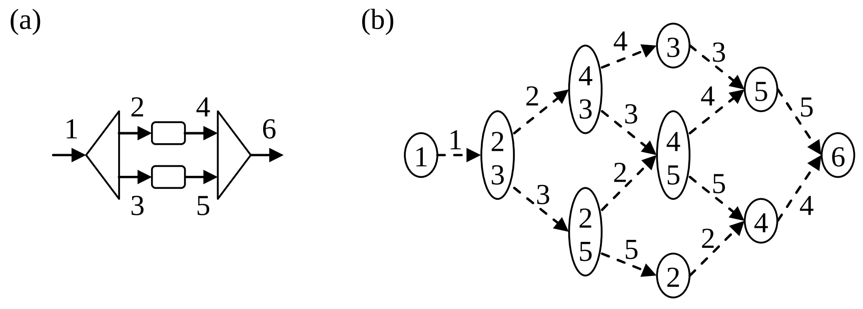


Figure 4.17: *(a) An unfolded flow graph and (b) its corresponding message-passing state graph. Edge labels identify the delivered message triggering the transition, and node labels indicate which messages are in transit.*

If we reach a single final state, we ensure that no message race or deadlock can occur for the given application input data. In our example, if a bug in the merge operation causes the content of the output message or the value of the local process variables to depend on the ordering of the delivery of messages 4 and 5, the final state will be different (Figure 4.18). When several final states are reached, we reveal the paths (i.e. the message orderings) leading to these states

[4]Although the data-driven acyclic graph construction of DPS applications makes deadlocks impossible on their own, they can be a manifestation of a prior message race (Section 4.2)

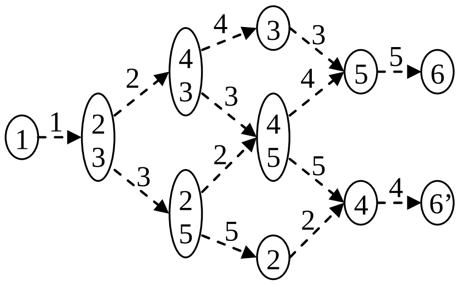


Figure 4.18: *Resulting state graph if the output of the merge operation is dependent on the ordering of its inputs (4 received before 5 vs. 5 received before 4).*

to enable their replay and study the erroneous execution.

Although any set of paths will do, we choose the ones with the longest common prefix in order to help the developer focus on the ordering variation that caused the divergence in the executions. This is achieved in two steps. We first identify within the message-passing state graph the shortest path P_f between two final states, such that the path follows a sequence of upstream links and a sequence of downstream links. In our example, this corresponds to the sequence of states $(6') \leftarrow (4) \leftarrow \binom{4}{5} \rightarrow (5) \rightarrow (6)$. The state S of P_f that has two outgoing links, i.e. the state $\binom{4}{5}$, splits P_f into P_{f1} and P_{f2}. The common prefix P_i between the diverging executions is given by the shortest path from the initial state of the message-passing state graph to S. The two paths with longest common prefix are then obtained by concatenating P_i with P_{f1}, and by concatenating P_i with P_{f2}. In Figure 4.18, one possible common prefix P_i is $(1) \rightarrow \binom{2}{3} \rightarrow \binom{4}{3} \rightarrow \binom{4}{5}$. The associated message orderings are then 1–2–3–4–5 and 1–2–3–5–4.

Message-passing state graphs have the benefit of taking local and global synchronizations into account. Figure 4.19a displays the unfolded flow graph of a two-iteration computation. For a single iteration, the unfolded flow graph graph accepts 6 orderings of length 6, i.e. testing all orderings requires delivering 6·6 messages. For two iterations, there are 36 orderings of length 11, which imply the delivery of 396 messages to execute all orderings. In contrast, the number of messages delivered to build the message-passing state graph is given by the number of edges in the graph and grows linearly, here from 13 to 26, with the number of iterations.

4.5.1 Reducing the number of visited states

The full message-passing state graph describes all possible orderings, without trying to eliminate the ones that are redundant. If we can determine *a priori* that different subpaths in the message-passing state graph will produce identical results, we may cut redundant branches by

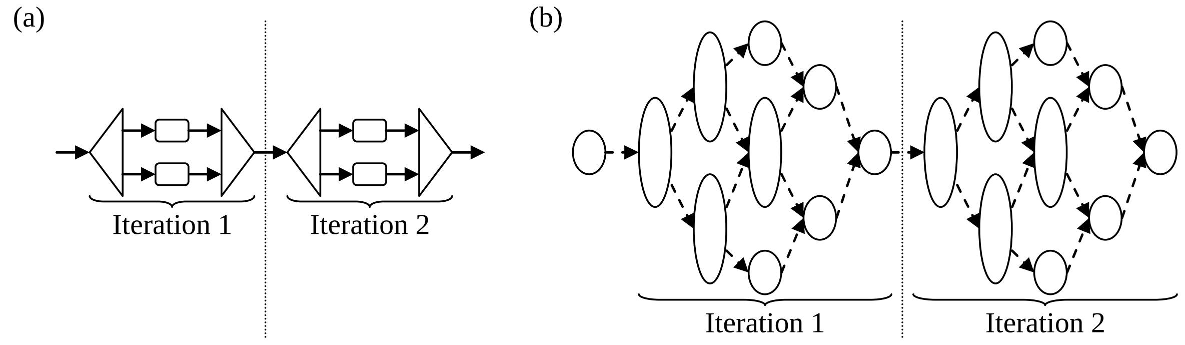


Figure 4.19: *The barrier synchronization caused by the merge-split sequence in the original flow graph (a) is reflected in the state graph (b).*

not sending all the messages that are in transit at a given state. Looking back at Figure 4.18 for example, sending only message 2 after the delivery of message 1 avoids testing all orderings where message 3 is delivered before message 2, and removes two states from the graph.

The atomic steps triggered by two messages a and b delivered to distinct threads do not directly interfere, i.e. one atomic step cannot modify the local thread variables read by the other atomic step. However, the future atomic steps triggered by a successor of b may interfere with the one triggered by a. If they do not, we may avoid delivering b; if they do, we have to deliver both a and b.

Detecting equivalent orderings and determining which messages we may avoid delivering at every state therefore requires *a priori* knowledge about future operations. Whereas the POEG contained the full information about the future behavior of the application, the dynamic analysis can only rely on information provided by the DPS flow graph. The flow graph specifies which operations may be triggered by a message and by its successors, as well as the thread collection on which these operations execute. However, it does not indicate exactly which thread will execute them.

Figure 4.20 displays an example based on the neighborhood-dependent application described in Section 4.2. Message 1 triggers operation C1, and one of its successors will eventually trigger an instance of operation E, which is a successor of C in the flow graph. However, since messages 1 and 2 are synchronized by operation D1, we do not need to consider E while determining the operations potentially interfering with messages 1 and 2. On the other hand, the first common successor of messages 1 and 3 is operation F1. Since the destination of each message is computed at runtime, the operation E1 triggered by a successor of message 1 may potentially be executed on the same process as operation C3. If this is the case, a race may appear if E1 modifies local process variables read by C3.

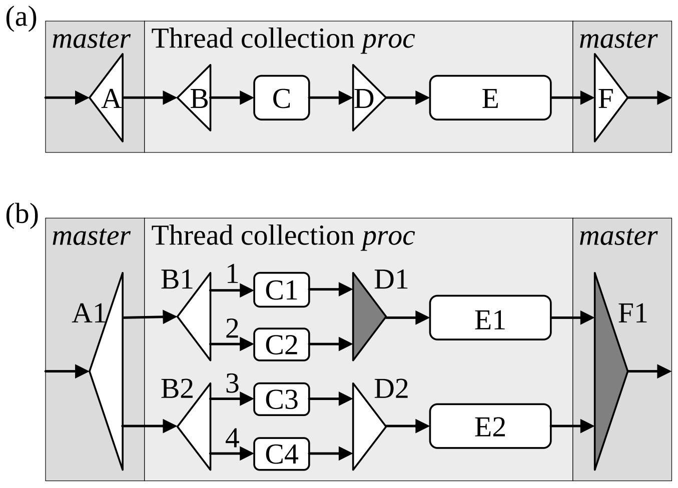


Figure 4.20: *(a) The flow graph of the application illustrated in Figure 4.2 and (b) its unfolded flow graph when deployed on two nodes. The first common successor of messages 1 and 2 is operation D in the original flow graph, while the first common successor of messages 1 and 3 is operation F.*

During the execution of the flow graph, its branches are identified thanks to the hierarchical construction of message identifiers (Section 2.5.1). The first common successor of two messages is therefore the merge operation that matches the split operation identified by the innermost split in the common prefix of their identifiers.

We may now establish rules for identifying sets of messages that may potentially interfere with each other. Let S be a state in the message-passing state graph with a set of messages in transit M. Let C_m and F_m be the set of *current*, respectively *future* interferers of a message $m \in M$.

1. C_m contains m and all messages $m' \in M$ such that m and m' are delivered to the same thread.

2. Given $m' \in M$, let *Succ*$_{m'}$ be the set of operations triggered by successors of m'. Then F_m contains all messages $m' \neq m$ such that the first common successor of m and m', or at least one operation of *Succ*$_{m'} \backslash$*Succ*$_m$ runs on the same thread collection as the operation triggered by m.

Both sets are computed dynamically for every message in transit of every state of the message-passing state graph. C_m contains the set of messages triggering atomic steps whose commutativity must be tested. Sending one message from C_m thus requires sending all the others. On the other hand, F_m contains the set of messages that must be sent before m, because they have successors that may interfere with m. Delivering these messages before m therefore enables reaching future states where these successors belong to C_m.

If the sets F_m and C_m associated to a message m are empty, the only message from S that we deliver is m. Otherwise, for each message $m \in M$ we augment its set F_m by recursively computing the union of F_m with the sets C_n and F_n for all $n \in F_m$. In other words, each message n in F_m has an associated set of current interferers C_n which must thus be sent together with the messages in F_m; similarly, the set F_n of messages that must be sent before n must also be included. The inclusion of current and future interferers must be repeated for every new message added to the augmented set F_m.

We then compare the augmented sets F_m associated to every message $m \in M$, and pick the set F_{m^*} with the smallest cardinality. We then deliver all the messages contained in F_{m^*}. The rationale for selecting the smallest set is that delivering fewer messages per state creates fewer branches in the message-passing state graph, which tends to reduce the number of states to be explored.

Messages that trigger read-only operations can be reordered freely without any impact on the computation result. The number of interfering messages in the sets C and F, and thus the number of messages to be sent from each state can thereby be further reduced if we know whether an operation only reads or modifies the local variables of the underlying process. Given such information and the sets C_m and F_m as defined above, we may remove from C_m and from F_m every message m' that is exchangeable with m as defined in Section 4.4.3, i.e. such that the atomic step triggered by m does not write or read a variable modified by the atomic step triggered by m'.

Further (although probably marginal) improvements could be achieved by using information about routing functions. For instance, DPS provides a built-in *ConstantRoute* routing functions that identifies the destination thread of a message by a constant. This particular route statically includes the message destination information into the flow graph. This may tell the validator about the precise location of operations beyond the ones triggered by the messages currently in transit: successor message known to trigger operations on distinct threads are guaranteed to not interfere.

The information about how operations read and write local thread variables is recovered as described in Section 4.4.5. However, computing the sets of future interferers of a message m requires identifying which operations may execute before the first common successor of m and of every other message in transit. Flow graph loops, flow control and stream operations all require special care:

- Loops repeating sequences of operations may cause an operation to become its own successor.
- Flow control between a split and a merge enables part of a split operation to be executed after parts of the matching merge operation[5]. A split may therefore become a successor of its matching merge, as well as of all the operations in between. Similarly, the enclosed operations become successors of the merge operation.
- We do not make any assumption about the internal message dependencies enforced by the stream operation: in contrast to Section 4.4, these dependencies may themselves depend on the ordering of messages. We therefore conservatively consider that streams never constitute the first common successor of two messages, and that any operation triggered after the stream operation may potentially be executed before operations placed before the stream in the flow graph.

A preprocessing phase before the start of the testing procedure therefore analyses the flow graph that is about to be executed and computes for each operation a list of potential successor operations that take these special cases into account. These updated successor lists are then used when determining the future interferers of a message.

4.5.2 Implementation

The implementation of the message-passing state graph construction and processing also builds on top of the simulator. Compared to the static POEG analysis method presented in Section 4.4, the checkpointing granularity is much finer. Both the threads and the operations must be serializable, and operations of tested applications must be restartable (Section 2.7.3). Like for messages and thread states, all fields must always be initialized; failure to do so leads to operation checkpoints that are different from all other checkpoints and needlessly prevents the aggregation of states within the message-passing state graph. The access vectors of operations are obtained by running the application once.

The message-passing state graph construction relies on all the features required to make a DPS application fault-tolerant, namely, the ability to checkpoint and restart threads and operations and the use of deterministic message identifiers [38]. An interesting side effect is that building the state graph also tests whether a DPS application tolerates faults.

[5]This feature is for instance used to improve the performance of the traveling salesman solver application in Section 3.6.

The testing procedure starts right before delivering the input message of the flow graph. We build the initial application state by taking a checkpoint of each thread and by making a copy of the input message. Unprocessed message-passing graph states are stored in a queue. For each unprocessed state S of the message-passing state graph, we determine the set of messages in transit that must be delivered. We then deliver one message from the set by pushing it into the incoming message queue of its destination thread, thereby triggering the associated operation. After the transition, we log the newly generated messages and checkpoint the thread to which the message was delivered. Together with the checkpoints of the other threads, this forms the new state of the application and a successor of S in the message-passing state graph. If the new state of the application has not been reached before, we add it to the queue of unprocessed states; otherwise we discard it. We then roll back the application to its former state S and deliver the next message. S is removed from the queue when all required messages have been delivered.

Application state checkpoints enclose the messages in transit (sorted by identifier), thread states (sorted by thread collection and index within the collection), and operations (sorted in the order of their creation). The ability to identify common checkpoints therefore relies on the uniqueness of message identifiers. Moreover, the hierarchical construction of identifiers enables messages at one point of the flow graph to have the same identifier even when the number of messages previously delivered may be different, thereby enabling states to be merged within the message-passing state graph (Figure 4.21).

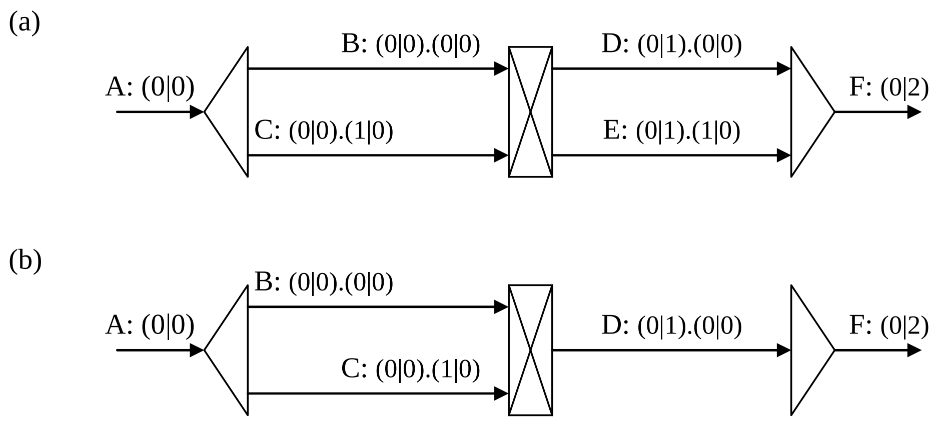


Figure 4.21: *(a) The stream operation sends two messages if it receives B before C, and (b) one message otherwise. The number of messages output by the stream operation has no influence on the message identifier of the output message F of the subsequent merge operation.*

4.5.3 Results

We now present practical results for our test applications. We use the same metric used for evaluating the POEG decomposition technique in the previous section, namely, the total number of messages that must be delivered to test all considered orderings. Like in Section 4.4.6, in case of a naive test of all possible executions, we compute the number of messages that must be delivered by multiplying the number of permutations by the number of messages sent during one execution. When using a message-passing state graph, the number of delivered messages corresponds to the number of edges in the graph.

Results for the neighborhood-exchange (NE) application of Section 4.2 are shown in Table 4.4. We compare the number of messages delivered for exhaustively testing two iterations of the neighborhood-exchange computation when naively executing all orderings, when using the full message-passing state graph, and when applying the optimizations described in Section 4.5.1. For two threads the optimized message-passing state graph reduces the number of messages that must be delivered by a factor of 10^{13} compared to the naive execution of all possible orderings.

Table 4.4: *Number of delivered messages for testing two iterations of the neighborhood-exchange application.*

	2 threads	4 threads	6 threads
All orderings	$5.6{\cdot}10^{16}$	–	–
Full state graph	1237	$3.4{\cdot}10^{6}$	–
Optimized state graph	843	$1.6{\cdot}10^{6}$	$4.2{\cdot}10^{9}$

Tables 4.5 and 4.6 summarize the results for the Floyd-Steinberg (FS) and the LU factorization applications. While testing the Floyd-Steinberg application on 4 threads, the optimizations described in Section 4.5.1 reduces the number of messages that must be delivered by a factor of 50 compared to the full state graph. Building the full state graph for more than 4 threads is not feasible due to the amount of memory required to store the application checkpoints. Within the LU application, the iterations of the computation are loosely synchronized in order to maximize the pipelining of the computation. Messages therefore have little dependencies between each other, causing the size of the message-passing state graph to explode, and almost canceling the benefits of the optimizations. Nevertheless, the use of the message-passing state graph drastically reduces the number of messages delivered for testing all orderings.

Finally, Table 4.7 presents results for the branch-and-bound TSP solver with 17 cities. As

Table 4.5: *Number of delivered messages for testing the parallel Floyd-Steinberg halftoning algorithm.*

	2 threads	4 threads	6 threads	8 threads
All orderings	$6.8{\cdot}10^8$	–	–	–
Full state graph	338	$3.9{\cdot}10^4$	–	–
Optimized state graph	47	765	$2.7{\cdot}10^4$	$1.0{\cdot}10^6$

Table 4.6: *Number of delivered messages for testing a pipelined parallel LU factorization.*

	3 threads	4 threads
All orderings	$>10^{17}$	–
Full state graph	4841	$6.2{\cdot}10^9$
Optimized state graph	4780	$6.2{\cdot}10^9$

explained in Section 3.6, messages distribute the value of the current best solution to processes in order to speed up the search, and a basic load-balancing scheme distributes computations more evenly among processes. Finding a good solution early may therefore reduce the number of messages and the amount of computations performed. This dependency of the content and destination of messages on the ordering of prior computations prevents the application of the POEG decomposition method. However, the variability of possible message contents also greatly increases the number of possible message-passing graph states. The running time therefore becomes prohibitive for testing the application exhaustively on more than two computation threads. All tests produce multiple final states, reflecting the existence of several solutions for our dataset: all final states showed the same minimum length for the total path, but with different orderings of cities.

Table 4.7: *Number of delivered messages for testing the traveling salesman solver.*

	2 threads
All orderings	$>10^{10}$
Full state graph	$8.1{\cdot}10^4$
Optimized state graph	$2.8{\cdot}10^4$

4.5.4 Scalability issues

While building the state graph, one often encounters the same messages and process checkpoints many times. Great care was therefore put into reducing the overheads caused by the management of the message-passing state graph. All messages and thread checkpoints are stored in hash tables to be quickly retrieved and compared. We use Paul Hsieh's aptly named *SuperFastHash* function [50] to minimize the time spent hashing states. When two hashes are equal, we avoid collisions by directly comparing the serializable objects. We then save memory by keeping a single physical copy of every element, and by discarding messages and checkpoints no longer needed. Also, since the delivery of a message may only modify the content of the thread that processed it, it is useless to checkpoint all threads after the delivery of each message. We therefore build the new application state by taking a new checkpoint for the destination thread of the delivered message, and by combining that checkpoint with the ones of the other threads from the previous thread state. These improvements reduce the testing time and memory consumption by several orders of magnitude.

The number of delivered messages impacts both the running time and the amount of memory required to build the message-passing state graph. The first two columns of Table 4.8 provides the regular application running time with four threads running on a single dual-core computer, and the memory occupied by the input data of the application (e.g. a matrix or an image). When building the message-passing state graph, we indicate the full running time of the testing procedure, and the amount of memory used by the testing process.

Scaling the tests from 4 to 6 threads leads to strongly increased execution times and memory consumption. These numbers need to be compared with the running time necessary to execute all orderings: on only two threads, we would already need in the order of 10^{14} seconds for

Table 4.8: *Running time [s] and memory consumption [MB] for regular executions and for tests relying on the message-passing state graph for the neighborhood-exchange (NE), Floyd-Steinberg (FS) and LU factorization applications.*

	Regular execution on 4 threads		Tests with message-passing state graph 4 threads		6 threads	
	[s]	[MB]	[s]	[MB]	[s]	[MB]
NE	0.13	2	452	227	117617	5689
FS	0.014	0.066	0.716	1.9	48	19
LU	0.014	0.115	33804	302	–	–

the neighborhood-exchange (NE) application, and 10^5 seconds for the Floyd-Steinberg (FS) application.

4.5.5 Testing a subset of orderings

Despite the orders of magnitude decrease in the number of explored states, using a message-passing state graph does not change the fundamental fact that the number of states and checkpoints grows exponentially with the number of threads. It is therefore crucial that we can generate a subset of orderings that reveals most if not all of the potential deadlocks and message races.

It is often the relative rather than the absolute ordering of two messages that causes a message race. For example, an error may occur for all orderings where message b is delivered before message a, regardless of the ordering of other messages. Many such errors occur when one message is unexpectedly delayed. We therefore want to produce a set of orderings likely to reveal existing races.

We generate a first reference ordering by delivering messages in the order in which they are sent by the application. This corresponds to a *breadth-first* traversal of the unfolded flow graph, and it is often close to the actual execution for applications where all threads have a similar behavior. We produce the ordering using a single FIFO queue: new messages are pushed at the back of the queue, and at each transition we deliver the message at the head of the queue (Figure 4.22b).

In order to fully change the ordering of the messages, we generate a second ordering equivalent to a *depth-first* traversal of the unfolded flow graph. This is achieved by sending all messages from a single branch before sending messages from another branch. This scheme simulates delays on specific branches, and is implemented as follows. Each state stores its messages in transit in a LIFO queue, or stack, of sets containing each all the messages generated by the reception of a single message. A transition is triggered by delivering one message from the set at the head of the stack. Once a transition completes, we copy the stack into the new state, remove the message that caused the transition and, if new messages were generated, push them as a new set on the top of the stack (Figure 4.22c). If no messages are produced, the head of the stack contains a set with unsent messages from a previous transition. When the head set contains more than one message, the choice of the message to deliver defines the order in which we execute the branches of the unfolded flow graph.

Each depth-first traversal corresponds to an ordering of branches. Different depth-first

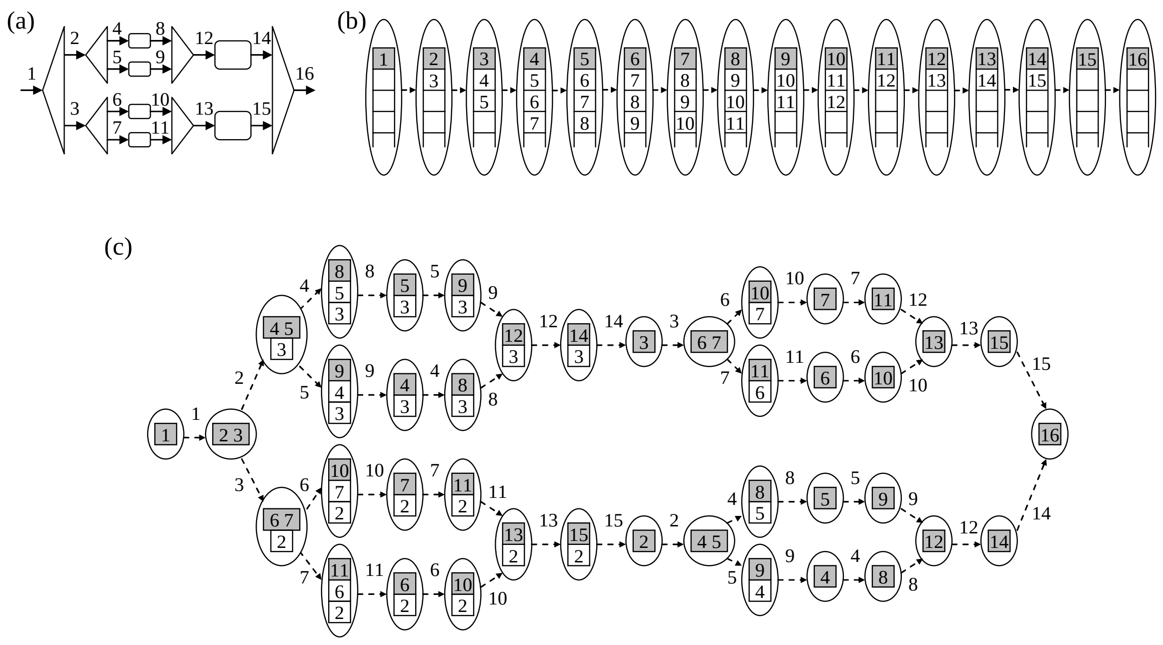


Figure 4.22: *(a) The unfolded flow graph of Figure 4.20; (b) the message-passing state graph for the breadth-first traversal; (c) the message-passing state graph of all the possible depth-first traversals (i.e. all branch orderings) of the unfolded flow graph. Highlighted numbers and edge labels represent the messages triggering the transition from a state to the next.*

traversals may be tested simultaneously by delivering more than one message from the head set. Figure 4.22c displays a message-passing state graph created by systematically sending all the messages in the head set of every state. Each path in that message-passing state graph represents a single depth-first traversal of the unfolded flow graph. The state graph describes all the possible orderings of branches within the unfolded flow graph of Figure 4.22a. Testing all branch orderings ensures that we test executions that maximize and minimize the delays that can be experienced by the operations on each branch.

Table 4.9 displays the number of messages delivered while testing all branch orderings of our applications unfolded flow graphs. We can see that for the FS application, testing all depth-first traversals requires the delivery of more messages than using the optimized state graph (Table 4.5). For less constrained applications such as the pipelined LU and the traveling salesman, testing all depth-first traversals strongly reduces the number of delivered messages.

These result are based on the full message-passing state graph. However, both the breadth-first and the depth-first traversals could be used after applying the partial-order reduction of Section 4.5.1. The effect of the partial-order reduction is to reduce the number of messages to be delivered in the head set of each state, thereby reducing the number of possible branch orderings to be explored.

Table 4.9: *Number of delivered messages [msgs], running time [s], and memory consumption [MB] for testing all unfolded flow graph branch permutations (i.e. all depth-first traversals) on 4 threads.*

	[msgs]	[s]	[MB]
Neighborhood-exchange	1737	8.32	10.0
Floyd-Steinberg	3513	1.92	4.6
LU factorization	$4.7 \cdot 10^5$	92	31.0
Traveling salesman	481	240	4.4

In practice, existing symmetries in the computations performed by different processes imply that multiple depth-first traversals can reveal the same error. In our tests, all errors were revealed using a few depth-first executions with distinct branch orderings.

4.6 Comparing the automated testing approaches

Both the static POEG decomposition and the dynamic message-passing state graph construction put a few requirements on the implementation of the application. We summarize them below and discuss the difficulty of applying them to an existing application. The first two requirements are specific to application testing. The other three are also required for making the application fault-tolerant.

Information about variables read and by operations. Providing read information to the validator requires replacing calls to access the local thread storage by a macro, which passes the name of the accessed variable to the validator (Section 4.4.5). Calls that fail to use the macro are detected and have the effect that the validator conservatively assumes that all variables are read by the calling operation, thereby decreasing the efficiency of the optimizations relying on that information.

Deriving read-write information from a single run. Both the static and dynamic testing methods assume that full read-write information can be obtained via an initial run of the application. For cases where that initial run is time consuming, or when it does not provide the complete access pattern of the operation, access vectors can be specified within a user-provided text file. Failure to implement this requirement reduces the efficiency of the optimizations.

Serializable thread states. Both the static POEG analysis and dynamic testing methods require the ability to checkpoint and recover the local storage of individual threads. This is a strict requirement since making thread states serializable also enables them to be compared automatically. In the vast majority of cases, this change simply consists in adding the various *ITEM* macros around the members of the local thread storage class (Section 2.7.2). Failing to make thread states serializable makes them look empty to the serializable object comparator, which then considers all thread states to be equal, and all operations to be read-only. Partial-order reductions using read-write information therefore considerably reduce the number of tested orderings, thereby hiding potential errors.

Developers may however decide to avoid serializing specific members of the thread state. Members that are not used directly by the computation, but for instance for debugging purposes, are ignored during state comparisons.

Deterministic identification of output messages of stream operations. Both automated testing methods rely heavily on the uniqueness and determinism of message identifiers. Stream operations are the only ones that may produce output messages in different orderings, and therefore with different identifiers. It is thus important that the developer uniquely identifies the messages using the extra parameter of the *postDataObject* function (Section 2.5.1). Failure to implement this requirement prevents the proper matching of messages during testing. When testing orderings within a POEG, incorrect matching of messages causes the validator to report spurious errors. When building a message-passing state graph, having different identifiers for the same message needlessly multiplies the number of states within the message-passing state graph.

Serializable and restartable operations. This last requirement is only a concern for building message-passing state graphs. Since we must be able to take and recover checkpoints of any possible application state, it is necessary that individual DPS operations may be checkpointed while they are suspended. This latter requirement is the most difficult to fulfill as its implementation is specific to each operation. Appendix A illustrates the transformation of the first split operation of the LU factorization application (operation (a) in Figure 3.9). Failure to implement this requirement prevents the proper restoration of suspended operations while recovering application states, and leads to incoherent application behavior as operations cannot maintain any internal state.

The two assumptions carried throughout this chapter are (1) that computations are deterministic and (2) that operations do not access non-read-only variables accessible from multiple DPS threads. Failing to make computations deterministic is immediately detected by the POEG testing method since every ordering produces a different results. While building the message-passing state graph however, non-deterministic computations prevent identifying identical states to be merged. The resulting graph is therefore likely to exhaust the available memory. Regarding our second assumption, writing to variables shared among different threads may remain undetected however, as these variables are not included in our application checkpoints. The effect of these variables can only be seen indirectly when they impact the content of thread states or messages, which are checked by the validator.

Provided that the above requirements and assumptions are fulfilled, both the static and dynamic testing methods are sound. They produce no false positives, in the sense that they only detect orderings that really produce different results. It is however up to the developer to determine whether these different results do or do not match the expected behavior of the application. When they test all orderings, the methods are also complete, i.e. they miss no race.

However, since the proposed methods only test an application for a single input dataset at a time, errors may remain for other datasets, particularly if these datasets trigger different execution paths within the application. Moreover, the fact that we partially re-execute the application code may induce very important testing times for long running computations. However, for three (NE, FS, LU) out of our four test applications and for many real-world codes, the number of messages sent and the communication patterns only depend on the problem decomposition granularity, and not on the size or content of the processed data. Testing small data sets is thus often sufficient for revealing message races and programming errors. Nevertheless, the results from the previous sections clearly point to the fact that only small portions of a parallel program can be tested exhaustively. One can for instance test orderings only between two barrier synchronizations.

For each of the static and dynamic testing method, we proposed two algorithms that trade completeness for the ability to test more of the interesting orderings. Theoretically, one may conceive applications whose message races or deadlocks are not exposed by the proposed orderings. We may therefore additionally execute randomly generated orderings in order to further reduce the probability that errors remain undetected. Similar approaches were successfully used for detecting data races in MPI [124] and multithreaded [88] applications. By increasing the number of randomly generated orderings, one can arbitrarily increase the confidence that no message race or deadlock exists.

4.7 Applying dynamic exploration to the static POEG decomposition

It is possible to combine the static and dynamic testing methods, by using a message-passing state graph to test the message orderings of subgroups decomposed using the POEG. Figure 4.23 illustrates how to test the subgroup (ii) of Figure 4.16c, which results from the full decomposition of a thread of the neighborhood-dependent computation.

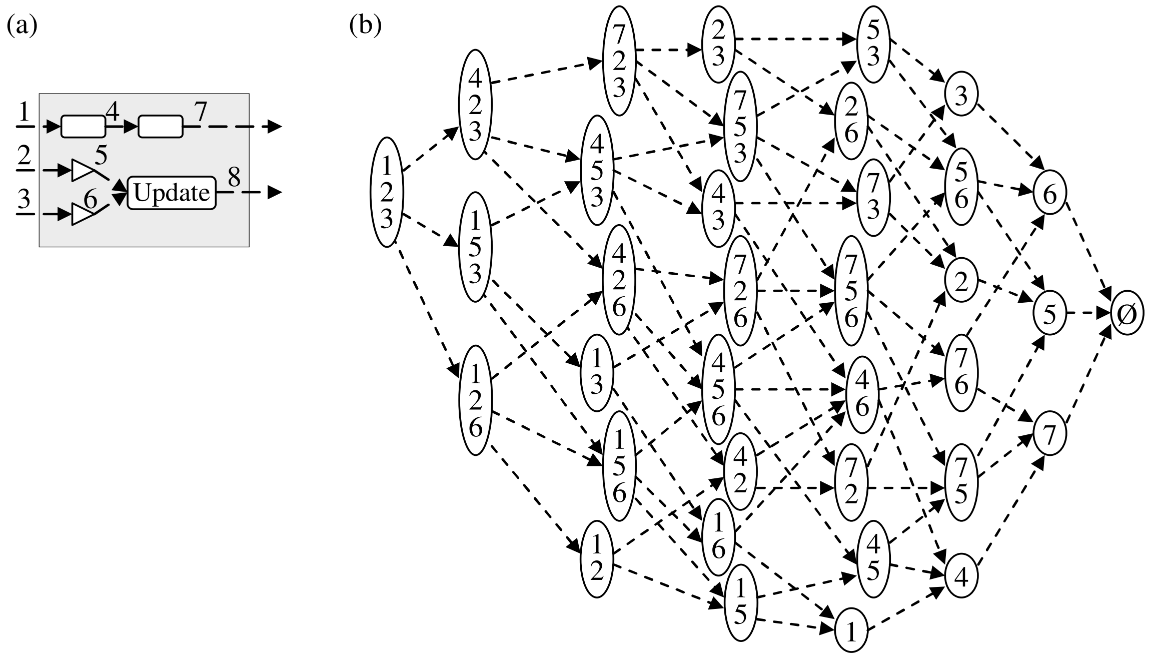


Figure 4.23: *(a) Subgroup produced after decomposing the POEG (subgroup (ii) of Figure 4.16c) and (b) its corresponding message-passing state graph. Testing the subgroup in (a) requires delivering 160 messages (8 · 20 orderings), while building the message-passing state graph requires the delivery of 75 messages.*

Such a combination enjoys the strength of both methods: it further reduces the number of orderings to be tested, consumes less memory than building the message-passing state graph for the whole application, and the different subgroups can easily be tested in parallel. However, it also combines their requirements: all the split, stream and merge operations of the application must be checkpointable and restartable, and memory is used for storing both the application trace and the message-passing state graph of the subgroup being tested. In addition, the combined two methods can only be applied to applications producing a fixed set of messages.

Nevertheless, the achievable reduction of the number of tested message orderings is very significant. Looking back at the best results achieved when decomposing the POEG of the neighborhood-dependent application (Table 4.1, page 110), it may be surprising to see that the number of delivered messages still grows exponentially with the number of participating threads, despite the fact that possible message orderings are tested independently for each thread. The explanation lies in the necessity to test the merge operations for all orderings of their inputs: a single merge operation with n input messages must be tested for $n!$ orderings, requiring the delivery of $n \cdot n!$ messages.

In practice however, operations are generally written such that the order of delivery of their input messages has no impact on their intermediate internal state. A same state can therefore be reached via many different orderings. Using a state graph to identify common intermediate states uses that property to drastically reduce the cost of testing the ordering of inputs of an operation.

Let us estimate the resulting number of messages to be delivered. If commuting any message pair does not change the computation outcome, the number of states reachable after the

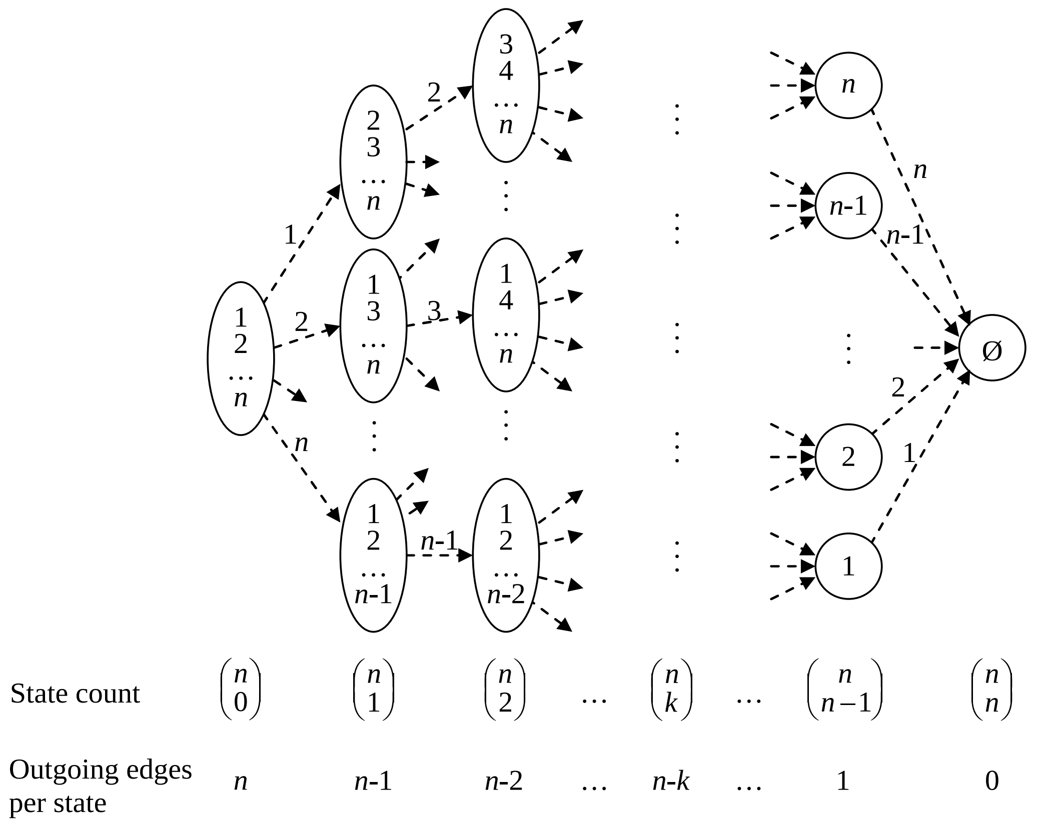


Figure 4.24: *Message-passing state graph produced when testing the orderings of the n inputs of a merge operation. The number of possible states after the delivery of k messages is $\binom{n}{k}$, and the number of edges for each of these states is $n - k$.*

delivery of k messages out of n is the number of combinations $\binom{n}{k}$, as illustrated in Figure 4.24.

The total number of reachable states is therefore

$$\sum_{k=0}^{n} \binom{n}{k}$$

Using the Binomial series $(1+x)^n = \sum_{k=0}^{\infty} \binom{n}{k} \cdot x^k$ and the fact that since n is a non-negative integer all terms of the sum are 0 for $k > n$, we obtain

$$\sum_{k=0}^{n} \binom{n}{k} = \sum_{k=0}^{\infty} \binom{n}{k} \cdot 1^k = 2^n \tag{4.3}$$

The associated number of delivered messages is given by the number of edges in the graph. After the delivery of k messages, the $n-k$ messages that remain to be sent from each of the $\binom{n}{k}$ states translate into $n-k$ outgoing edges (Figure 4.24). The total number of edges in the graph is therefore computed as

$$\sum_{k=0}^{n} (n-k)\binom{n}{k} = \sum_{k=0}^{n} \frac{n!}{(n-k-1)!k!} = n\sum_{k=0}^{n} \binom{n-1}{k} = n\sum_{k=0}^{n-1} \binom{n-1}{k}$$

Using the result of equation 4.3, the number of edges is therefore $n \cdot 2^{n-1}$, or $\frac{n}{2}$ times the number of states in the graph. Although the general complexity remains exponential, the reduction is significant, as shown in Table 4.10. The benefits of the reduction more than compensates for the cost of taking and recovering the checkpoints.

Table 4.10: *Number of delivered messages for testing a single merge or stream operation.*

Number of input messages (n)	4	6	8	10
All orderings ($n \cdot n!$)	96	4320	$3.2 \cdot 10^5$	$3.6 \cdot 10^7$
State graph ($\frac{n}{2} \cdot 2^n$)	32	192	1024	5120

We estimate the effect of the combined method for the NE, FS and LU test applications on the number of messages to be delivered for testing these applications. We obtain these numbers by computing the message-passing state graph of each subgroup of messages produced by the POEG decomposition of Section 4.4. The message-passing state graph is produced from the partial-order execution graph assuming that all operations are commutative, which is correct for our sample applications. We recall the best values obtained with the static POEG decomposition (Section 4.4) and dynamic message-passing state graph construction (Section 4.5) in order to compare the results.

Table 4.11: *Number of messages to be delivered for testing the whole application using a POEG, a message-passing state graph and by combining the two.*

NE	2 threads	4 threads	6 threads	8 threads
Static decomposition	860	1932	$1.3{\cdot}10^4$	$7.3{\cdot}10^5$
Message-passing state graph	843	$1.6{\cdot}10^6$	$4.2{\cdot}10^9$	–
Combined	142	326	774	2566

FS	2 threads	4 threads	6 threads	8 threads
Static decomposition	42	1280	$7.4{\cdot}10^4$	$6.8{\cdot}10^6$
Message-passing state graph	47	765	$2.7{\cdot}10^4$	$1.0{\cdot}10^6$
Combined	25	185	1381	9265

LU	3 threads	4 threads	5 threads
Static decomposition	$7.9{\cdot}10^5$	–	–
Message-passing state graph	4780	$6.2{\cdot}10^9$	–
Combined	382	$3.5{\cdot}10^4$	$1.4{\cdot}10^7$

The neighborhood-dependent computation (NE), strongly benefits from the PEOG decomposition thanks to its well defined synchronizations. As mentioned above, its scalability is however limited by the requirement to test orderings of merge operations: on 6 threads, testing a single merge operation requires the delivery of $8{\cdot}8! = 3.2{\cdot}10^5$ messages. Using message-passing state graphs for testing the subgroups leads to an overall reduction in delivered messages of more than two orders of magnitude on 8 threads.

Reductions are even more significant for the Floyd-Steinberg error diffusion algorithm. Its implementation uses stream operations to loosen the synchronizations between the different execution stages, which reduces the efficiency of the static POEG decomposition. The use of the dynamic method to test orderings within subgroups of messages therefore produces good improvements. The same comment applies for the LU factorization application: while the total number of message orderings within subgroups is too large to even count them, the number of delivered messages remains tractable when the POEG decomposition and message-passing state graph building are combined.

4.8 Potential improvements

Apart from further optimizations to improve the identification and prevention of equivalent message orderings, several additional improvements may be achieved through additional combinations of the features described so far.

4.8.1 Optimizing operation scheduling

Section 3.8.1 mentioned the possibility of adding noise to the duration of communication times while simulating application executions. This hints at another possible development: find an operation schedule, using operation dependencies to determine possible orderings, that minimizes the application running time. This could also help identify which merge-split sequences would be advantageously replaced by stream operations.

4.8.2 Using checkpoints with manual testing

Manually testing an application generally involves replaying multiple times slightly varying executions. Enabling the developer to take checkpoints of his application may add significant value to the debugger. By taking global snapshots of the current state of the application, the developer may roll back to it later without reexecuting the whole application. Multiple snapshots could be differentiated using thumbnails of their respective flow graph views. Combined with the ability to reorder and modify messages, this feature would enable interactively testing multiple execution scenarios within a specific part of the whole application. The debugger may then retrieve the thread states after the execution of each ordering and automatically compare and highlight differences between the final states to reveal message races.

Since the flow graph description provide the debugger with a full knowledge about the causality between the executed operations, it would also be possible to undo a specific operation and determine which causally dependent operations must also be undone to maintain a consistent state, thereby providing a finer grain of control while stepping back to previous execution states.

4.8.3 Combining execution visualization and POEG decomposition

One advantage of the static POEG decomposition method is that errors are detected as soon as they occur. Since the message subgroups are statically created from the original execution of the application, these subgroups can be represented graphically by highlighting the associated operations within an unfolded flow graph representation. Such a view is already provided by the DPS debugger presented in Section 4.3. Extending it would enable the validator to highlight the different subgroups within the debugger's user interface, and enable the developer to decide which subgroups should be tested, and whether tests should be exhaustive or performed using a heuristic. Color codes may then be used to represent each subgroup's test results (e.g. *skipped*, *success* or *fail*), as well as messages or local thread storage that have been found to be different for specific orderings of messages.

Regarding the ability to interactively reorder messages within incoming message queues of the threads, the number of possible orderings explodes when the number of messages is large. However, the partial-order reduction of Section 4.4.3 could be applied in order to provide the developer with a shorter list of significant orderings.

4.8.4 A note on non-deterministic computations

Our assumption that computations must be deterministic makes it easier to describe and think about the testing problem. However, all the notions of message, thread state and application state equality are based on the comparison of serializable objects. A developer willing to write custom comparison methods for his objects may thus loosen the determinism requirements. One could for instance imagine defining equivalence functions that compare some statistic of the object content, such as "two random vectors are equal if the mean of their components differ by less than 1%".

As a final comment, we should note that making computations deterministic by using a fixed seed once and for all at the beginning of the computation may be unsufficient when multiple operation instances draw random numbers. Accessing a global pseudo-random number generator (PRNG) from all threads causes the allocation of random numbers to operations to depend on the ordering of the operations. The same problem applies if each thread uses its own PRNG: as multiple operation orderings will cause random values to be drawn in different orders, the results of the computation will be different.

4.9 Conclusion

We presented three methods for inducing and detecting message races within DPS applications. As a first approach, we integrated debugging hooks within the DPS library and implemented a user interface for visualizing the execution of DPS applications. By dynamically drawing the application flow graph as it unfolds, the debugger enables the programmer to easily see the state of the execution as well as the status of every thread and operation. General features like operation and message breakpoints, operation inspection using a sequential debugger and message queue inspection provide the developer with much insight. The ability to influence the application through the reordering or modification of messages provides the developer with full control over the execution of the application. This control can be used to run executions that occur only rarely in practice and compare their outcomes.

Our second approach models a parallel application as a Partial Order Execution Graph (POEG). We partition the POEG into smaller parts, firstly by distinguishing the sets of messages triggering computations in different memory spaces, and secondly by separating causally dependent message subsets. Leveraging information about how the computations triggered by each message read or modify local state variables, we identify equivalent orderings within each subset. Equivalent orderings are prevented by adding edges to the POEG, which force the relative delivery order of the messages. In order to further reduce the number of orderings to be tested, we then propose solutions for generating a subset of orderings that are still able to reveal many potential errors.

We integrated the POEG decomposition and race detection methods within the DPS parallelization framework, where POEGs are easily derived. For three different parallel applications, we evaluated the influence of the proposed techniques on the total number of messages that must be delivered to the application for testing all orderings. In all cases, the number of orderings that must be tested is reduced by several orders of magnitudes. Since this approach is based on a static graph analysis, it cannot be applied to applications that may produce different sets of messages depending on the evolution of the computation.

This limitation is overcome by our third approach, which dynamically tests multiple message orderings. We represent the multiple orderings using a message-passing state graph built at runtime, which ensures that parts common to multiple orderings are executed only once. We then use information about future computations and about how these computations read or write local process variables to identify equivalent orderings within the message-passing state graph. One drawback of this method is the excessive amount of memory consumed to store the message-passing state graph. We therefore also show how to generate a subset of orderings that are still able to reveal most errors.

We also integrated the dynamic message-passing state graph construction and analysis method within DPS, and leverage its support for checkpointing and restarting individual threads during a computation. For four different parallel applications, we evaluated the influence of the proposed techniques on the total number of messages that must be delivered in order to test all orderings. The message-passing state graph and the described optimizations enable reducing the number of delivered messages by many orders of magnitude compared to the naive approach. The proposed partial tests revealed all errors present in our experiments. We finally described further potential improvements obtained by the combination of the approaches.

All the techniques described in this chapter were presented in the context of DPS applications. They have been published in [3, 97, 101]. The next chapter presents a partial generalization of these methods to MPI applications using a subset of MPI calls.

Chapter 5

Using MPI

5.1 Introduction

The previous chapters presented means for predicting the running time and detecting synchronization errors within parallel applications written using the Dynamic Parallel Schedules library. In practice however, the vast majority of message-passing parallel applications rely on libraries implementing the Message Passing Interface (MPI) standard [110].

MPI is a standard developed through an open process that currently involves more than 40 industrial companies, research laboratories and academic institutions. The first version of the standard was published in 1994 (MPI-1.0), and new functionalities were standardized in 1997 (MPI-2.0). Newer documents include clarifications and errata corrections and are referred to as MPI-1.3 and MPI-2.1.

Even though other contenders such as Parallel Virtual Machine (PVM [112]) offered parallelization capabilities, MPI has now become the *de facto* choice for writing message-passing parallel applications. One of its strengths is that it appears to have found the proper level of abstraction by handling all the low-level details of establishing and performing communications without putting any restriction on the type of communication patterns that may be implemented. Moreover, although the API has now grown to more than 200 functions, about ten to twenty functions suffice for the vast majority of applications. This relative simplicity also contributes to the appeal of MPI. Quality open-source MPI implementations such as MPICH2 [115] and OpenMPI [34] are available for all major operating systems, and vendors often provide optimized libraries for their particular hardware. These two features guarantee the portability of MPI applications to many types of hardware.

The present chapter presents an adaptation to MPI of the work described in the previous chapters. This generalization only considers features from version 1 of the MPI standard. Features introduced in MPI-2 such as dynamic process instantiation, one-sided communications and parallel I/O are not supported.

Section 5.2 starts by briefly presenting the major concepts behind MPI, as well as the most commonly used functions of MPI-1. We also show how complex C++ objects can be transfered using MPI thanks to the *autoserial* library [99]. We then illustrate potential errors that may occur within MPI applications in Section 5.3. Section 5.5 adapts the interactive testing methods and graphical tool presented in Section 4.3 to MPI applications. The static POEG decomposition and the dynamic construction of the message-passing state graph are applied to MPI applications in Section 5.5, and Section 5.6 summarizes the results.

5.2 MPI: the Message Passing Interface

MPI applications are started through an application launcher (*mpiexec*), which starts multiple copies of the same executable on a set of compute nodes. The list of available nodes is typically specified within a configuration file. Before performing any communication, every process must call the *MPI_Init* function to set up the MPI runtime. By default, the MPI library assumes that all processes are single-threaded. An extension offered by MPI-2 enables application to determine whether the underlying implementation is thread-safe using the *MPI_Init_thread* initialization function. This function may return *MPI_THREAD_MULTIPLE* to indicate that multiple threads may simultaneously perform communications. Before exiting, every process must call *MPI_Finalize* to clean up the state of the MPI library.

MPI uses the notion of *communicators* to specify sets of processes that may communicate with each other. A communicator has a *size*, defined by the number of processes that it contains. Within a communicator, each process is identified by its *rank*, i.e. a number between 0 and $n-1$ where n is the communicator size. A process may belong to multiple communicators, and may have a different rank within each communicator. We assume in the current chapter that the set of processes remains fixed during the lifetime of a communicator[1].

A special communicator, *MPI_COMM_WORLD*, is initialized by the MPI runtime and contains all the processes started by the MPI launcher. The size of *MPI_COMM_WORLD* indicates how many processes participate in the computation. MPI provides functions that a process may

[1]While this limitation has been removed in the version 2 of the MPI standard, very few applications support varying numbers of processes in practice.

Listing 5.1: *The basic structure of a simple compute farm parallel application.*

```
int main(int argc, char *argv[])
{
  int rank, nProcs;
  MPI_Init(&argc, &argv);
  /* Get process rank in MPI_COMM_WORLD */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  /* Get the size of MPI_COMM_WORLD, i.e. the total no. of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
  printf("I'm process %d out of %d\n", rank, nProcs);

  if (rank == 0) // Process 0 is the master
  {
    /* Distribute tasks to the workers */
    ...
  }
  else
  {
    /* Receive tasks from the master and perform computation */
    ...
  }
  MPI_Finalize();
  return 0;
}
```

use to determine its rank within a communicator and the size of a communicator.

Listing 5.1 illustrates the basic structure of a simple master-slave compute farm application. After calling *MPI_Init*, each process retrieves its rank within the *MPI_COMM_WORLD* communicator as well as the number of instantiated processes. Although each copy of the program executes the same code, the value of the *rank* variable is different for each process. Developers may therefore use that information to assign different roles to different processes. In this example, process 0 is the master responsible for distributing the tasks and collecting the results, while the other processes are the workers and perform the actual processing. Different processes may thereby perform different computations.

Sending tasks to the worker processes and receiving the computation results requires some communications between the processes. The following sections briefly presents the small subset of MPI concepts and calls most commonly used within MPI applications. We refer the reader to the numerous resources available on the web (e.g. [77]) as well as in print (e.g. [110, 41]) for further details.

5.2.1 Point-to-point communications

A *point-to-point* communication enables sending a message from a sender process to a receiver process. The key requirement is that both the sender and receiver process must participate in the communication by calling matching *send* and *receive* functions.

Listing 5.2 displays an example of how the master process of rank 0 may send a message to each of the other processes. The sender calls the *MPI_Send* function once for each message it wants to send, while each receiver calls the *MPI_Recv* function to receive the message. The parameters of the *MPI_Send* call specify which buffer of data and how many elements within that buffer are to be sent. It must also specify the type of these elements, the rank of process that will receive the message (here it is represented by *dest*), a tag that can be used to identify this communication, and the communicator in which the destination rank should be found. The receiver must also specify a buffer, its length and the type of data it expects, the source of the message (here it is 0), the tag, the communicator, and a *status* parameter.

While this example illustrates the simplicity of sending a message between processes, it

Listing 5.2: *Simple send of memory buffer from process 0 to process 1.*

```
/* Initialize MPI, rank, nProcs and an array of buffers of integers */
...

if (rank == 0)
{
  /* Send tasks */
  for (int dest=1; dest < nProcs; ++dest)
    MPI_Send(bufs[dest-1], bufSize, MPI_INT, dest, tag, MPI_COMM_WORLD);
  /* Receive results */
  for (int src=1; src < nProcs; ++src)
  {
    MPI_Status s;
    MPI_Recv(res[src-1], resSize, MPI_INT, src, tag, MPI_COMM_WORLD, &s);
  }
}
else
{
  MPI_Status s;
  /* Receive task from process 0 */
  MPI_Recv(taskBuffer, taskSize, MPI_INT, 0, tag, MPI_COMM_WORLD, &s);
  ...
  /* Send result to process 0 */
  MPI_Send(resultBuffer, resultSize, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
MPI_Finalize();
```

also reveals several major differences with the behavior of DPS. Firstly, the message must be expected by the destination for the transmission to complete. The parameters of the send and the receive calls must match: the tag and the communicator must be strictly equal for the send to succeed, and the size of the message must be known by the receiving process. If a master process must send multiple messages to each slave, each slave must know how many messages to expect in order to perform the right number of receive calls.

Secondly, in this example the master process remains idle while waiting for the slaves to complete their computations. The order in which messages are received is also enforced by the *src* parameter of the receive call of the master (line 13). Load-balancing, buffering at the destination and flow control must be programmed explicitly, which has a significant impact on the code complexity.

Blocking point-to-point communication functions

MPI specifies both blocking and non-blocking types of point-to-point communication functions, which provide different synchronization semantics.

A call to the *MPI_Send* function returns when the buffer specified as a parameter can be safely used again by the application. This either means that the message has been sent, or that the buffer content has been copied within the MPI library. The internal copy is allowed for performance reason. However, since the size of the behavior of the actual buffering is specific to the MPI implementation, applications should not rely on it: a functional program may see errors occurring when the size of the sent buffer increases or when another MPI implementation is used.

The *MPI_Ssend* function, or synchronous send, provides stricter guarantees. The function may return only when the buffer may be reused by the application and the matching receive has been posted by the destination process.

The *MPI_Recv* function is the only blocking receive. Calls to that function block until the message has been copied within the specified reception buffer.

Non-blocking point-to-point communication functions

MPI also provides a non-blocking (or *instantaneous*) version of every blocking send and receive functions. The ones corresponding to the blocking functions described above are *MPI_Isend*, *MPI_Issend* and *MPI_Irecv*. Non-blocking functions return immediately, and take one extra parameter in the form of an *MPI_Request* structure. This structure is filled by the MPI runtime and acts as a request identifier.

Request identifiers must be used with *MPI_Wait* or *MPI_Test* functions to query the MPI runtime for the completion of the non-blocking calls. *MPI_Test* always returns immediately and sets a flag to indicate whether the call associated to the specified *MPI_Request* structure has completed or not. *MPI_Wait* blocks until the call has completed. Calling *MPI_Isend*, resp. *MPI_Issend* immediately followed by *MPI_Wait* is identical to calling *MPI_Send*, resp. *MPI_Ssend*, while calling *MPI_Irecv* followed by *MPI_Wait* is identical to calling *MPI_Recv*.

Wait and test functions with *_all*, *_any* and *_some* suffixes enable querying or waiting on multiple requests simultaneously.

Wildcards

Two special constants, called wildcards, can be used to relax the set of sends that can be matched by a receive call. The first is *MPI_ANY_SOURCE*, which removes filtering based on the sender of the message. Due to the lack of synchronization between the processes taking part in the computation, using *MPI_ANY_SOURCE* may make the application execution non-deterministic. This may for instance induce non-deterministic deadlocks as well as message races. Section 5.3 shows several examples of such synchronization errors.

The second wildcard, *MPI_ANY_TAG*, removes filtering based on the tag used in the send call. The MPI standard mandates that communication channels must not reorder messages during transmission between a single pair of nodes. In other words, if process p_i sends a message A and then a message B to process p_j, then a receive call at p_j that matches both messages A and B will receive message A. This FIFO property has the consequence that using *MPI_ANY_TAG* does not make the application non-deterministic, provided of course that the specified message source is not *MPI_ANY_SOURCE*. Indeed, the receive call will always match the first matching message received on the same communicator.

When using *MPI_ANY_TAG* or *MPI_ANY_SOURCE*, the actual tag or source of the matched send is stored within the *MPI_Status* parameter of the receive call. The application may thereby perform specific processing based on the matched tag or source.

In this chapter, we refer to *wildcard receives* to indicate receives that may induce non-determinism. Since *MPI_ANY_TAG* induces no non-determinism, wildcard receives only refer to calls using *MPI_ANY_SOURCE* regardless of the tag that they use.

5.2.2 Collective communications

Collective communications provide simple ways to perform communications that involve all the processes within a single communicator. Collective communications require the participation of all processes in the communicator in order to avoid deadlocks, i.e. all processes must call the same collective communication function. Collective communications can be described as *fan-in*, *fan-out*, or *all-to-all* communications.

Fan-out collective communications

Fan-out collective communications are *rooted*. One process, the root, sends data, and all processes receive data. The simplest call is *MPI_Bcast*, which broadcasts a buffer to all processes. Listing 5.3 illustrates its use.

Listing 5.3: *Use of* MPI_Bcast *to broadcast data to all processes. Process 0 is the root of the broadcast.*

```c
int root = 0; /* Process 0 is the root */
int *buffer = (int*)malloc(bufferSize * sizeof(int));
if (rank == root)
{
  /* Fill in buffer */
  ...
}

/* Each process calls the broadcast function. The buffer of the */
/* root process is broadcasted to all processes                */
MPI_Bcast(buffer, bufferSize, MPI_INT, 0, MPI_COMM_WORLD);

/* After the call, the buffer variable of each process */
/* has the same content.                               */
```

The other fan-out collectives are *MPI_Scatter* and *MPI_Scatterv*, which split a buffer into multiple pieces of the same, respectively of different sizes. Each participating process (including the root) receives one piece of the originating buffer.

Fan-out calls may be buffered at the root process, i.e. the call may return before all processes have called the collective communication function. Non-root processes block until the data has been received and copied into their receive buffer.

Fan-in collective communications

Fan-in collectives are also rooted, but the root process is the receiver of the data sent by itself and by other processes. The fan-in collective calls are *MPI_Gather*, *MPI_Gatherv* and

Listing 5.4: *Use of* MPI_Gather *to collect the ranks of all processes. Process 0 is the root of the collective.*

```c
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int *buf;
  if (rank == 0)
    buf = (int*)malloc(size * sizeof(int));

  /* buf only needs to be allocated at the root, i.e. at process 0 */
  MPI_Gather(&rank, 1, MPI_INT, buf, 1, MPI_INT, 0, MPI_COMM_WORLD);

  /* The root process received the rank from each process */
  /* in MPI_COMM_WORLD                                     */
  MPI_Finalize();
  return 0;
}
```

MPI_Reduce. The gather functions collect smaller buffers from each process into a larger one at the root. The reduction function aggregates buffers from all processes by applying a commutative operation such as “sum” on all buffer elements. Listing 5.4 shows how *MPI_Gather* may be used to collect the ranks of all processes within a buffer at the root. For fan-in collectives, processes other than the root may buffer the data to be sent, i.e. the function may return before all the processes have entered the collective.

All-to-all collective communications

All-to-all collective communication functions require each participating process to both send and receive data. These functions are therefore synchronizing, in the sense that every process must have called the appropriate collective function before the function may return on any process. These functions include *MPI_Alltoall*, *MPI_Allreduce* and *MPI_Allgather*. For instance, *MPI_Allgather* is a variant of *MPI_Gather* where all processes receive the result of the gather operation.

One special all-to-all collective function, *MPI_Barrier*, does not send or receive any data, but its sole purpose is synchronization. A call to *MPI_Barrier* does not return until all processes of the communicator have entered the barrier.

As for point-to-point communications, collectives require that all participating processes call matching functions with matching parameters. While parameters are specific to each function, all processes must agree on the communicator used, and all rooted functions must use the same root for the communication to complete successfully.

5.2.3 The MPI profiling interface

The MPI standard provides support for developers of MPI tools through the *profiling interface*. Each MPI function has two prototypes, one with a *MPI_* prefix, and the other with a *PMPI_* prefix. Both prototypes have identical signatures. The presence of two prototypes for the same function enables tool developers to reimplement the base *MPI_* function while still being able to call the *PMPI_* version in order to use the functionality of the MPI library. Once the desired functions have been reimplemented, typically within a separate library, any application can link that interception library in addition to the MPI library. While running, the application then calls the interception functions instead of the ones provided by the MPI library. Listing 5.5 shows how the *MPI_Send* and *MPI_Finalize* functions can be reimplemented to provide statistics about the number of sends performed.

Other techniques can be used to intercept and reimplement library calls. One possibility is to use a library stub that defines wrappers around the functions to intercept, while making sure that only the wrapper functions are exported. An interception library may then reimplement the

Listing 5.5: *Using the MPI profiling library to count the number of calls to* MPI_Send. *The count is displayed when the process calls* MPI_Finalize.

```
static int sendCallCount = 0;

int MPI_Send(void *buf, int count, MPI_Datatype type,
             int source, int tag, MPI_Comm comm)
{
  sendCallCount++;
  /* Call the actual send function */
  return PMPI_Send(buf, count, type, source, tag, comm);
}

int MPI_Finalize()
{
  int rank;
  PMPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Process %d called MPI_Send %d times\n", rank, sendCallCount);
  /* Call the actual finalize function */
  return PMPI_Finalize();
}
```

Listing 5.6: *Intercepting calls to* MPI_Send *from Fortran programs. The function is reimplemented in C and must convert Fortran types into C types.*

```c
/* The C function name associated to a Fortran function has a
   trailing _ */
void mpi_send_(void *buf, int *count, MPI_Fint *type, int *dest,
               int *tag, MPI_Fint *comm, MPI_Fint *ierr)
{
  /* Convert datatype and communicator to C types */
  MPI_Datatype c_type = MPI_Type_f2c(*type);
  MPI_Comm c_comm = MPI_Comm_f2c(*comm);
  /* Call the C wrapper function */
  *ierr = MPI_Send(buf, *count, c_type, *dest, *tag, c_comm);
}
```

original functions, and use the wrappers to call the original implementation. For dynamically loaded libraries, it is also possible to explicitly load the desired functions, e.g. via the *dlopen* and *dlsym* functions on Unix. This technique is for instance used by Jockey [93], a record/replay library for C programs that intercepts calls to the *libc* standard C library. However, both solutions involve platform- and compiler-specific flags and functions and require the developer to reimplement every single library function used by an application. In contrary, the profiling interface provided by MPI has the enormous advantage of providing a cross-platform solution enabling developers to reimplement only a subset of MPI functions without restricting the set of functions that may be called by the application.

The MPI standard specifies language bindings for C, C++ and Fortran. Fortran calls can also be intercepted by using slightly different function prototypes. Communicators and datatypes are represented differently in Fortran and in C. The standard therefore provides functions such as *MPI_Comm_f2c* and *MPI_Comm_c2f* to perform conversions between the two representations. Listing 5.6 shows the C implementation of the Fortran prototype for the *MPI_Send* function. The interception function calls *MPI_Send* rather than *PMPI_Send* so that Fortran calls are processed identically to C calls.

5.2.4 Sending and receiving complex C++ objects

All MPI calls are designed to transmit buffers (or arrays) of elements of MPI types. In addition to basic types such as *MPI_INT*, *MPI_DOUBLE* or *MPI_BYTE*, MPI provides ways to define custom datatypes, consisting for instance of vectors of basic type elements or of simple C struct. However, custom datatypes cannot be used to represent pointers and complex data structures such as linked lists or trees, or C++ STL containers.

Listing 5.7: *Sending complex C++ objects using the* autoserial *library.*

```
/* Object declaration */
class A : public ISerializable {
  AS_CLASSDEF(A)
  AS_MEMBERS
    AS_ITEM(int, a)
    AS_ITEM(double, b)
    AS_ITEM(std::vector<int>, v)
  AS_CLASSEND;
public:
  A() { a=2; b=3.14; v.push_back(10); v.push_back(29); }
};

int main(int argc, char *argv[]) {
  /* ... */
  if (rank == src) {
    A a;
    AS_MPI_Send(a, dest, tag, comm);
  } else if (rank == dest) {
    A *a;
    AS_MPI_Recv(a, src, tag, comm, status);
    delete a;
  }
  /* ... */
}
```

Sending such complex data structures therefore requires the developer to explicitly write the serialization and deserialization code. We therefore added MPI wrappers within the *autoserial* library [99] that enable sending and receiving complex C++ objects. Other approaches propose MPI extensions to send complex objects by relying on programming languages providing built-in serialization capabilities such as C# and Java [79, 126] or by requiring developers to write their own serialization code [56, 68].

The *autoserial* library currently only includes wrappers for the *MPI_Send* and *MPI_Recv* functions. Listing 5.7 shows the use of the wrappers to send a simple object composed of an integer, a double and an STL vector containing integers. Appendix B provides performance results for several types of serializable objects.

5.3 Synchronization errors in MPI applications

The requirement that all processes involved in a communication must explicitly participate by calling a specific function with specific parameters enables many potential errors such as deadlocks and message races. This section briefly illustrates a few error cases. In the figures below, each box represent the call of an MPI function. In order to improve their readability, MPI functions are displayed without their *MPI_* prefix. The single parameter of each function indicates the rank of the destination, source or root process depending on whether the process calls a send, receive or collective function. We use the source '*' to represent wildcard receives.

Listing 5.8 sketches the code of a program sending a single message from process 0 to process 1. However, the program assumes that only two processes will be started; one sends the message and the other receives it (Figure 5.1a). If three processes are started for instance, both process 1 and process 2 will call the *MPI_Recv* function. Since process 0 only sends a single message to process 1, the receive call from process 2 will never return and cause a deadlock (Figure 5.1b). The obvious solution in this example is to replace the single *else* by *else if (rank == 1)*.

Listing 5.8: *Send of a memory buffer from process 0 to process 1.*

```
/* Initialize MPI, rank and buffer of integers */
/* ... */

if (rank == 0)
  MPI_Send(buffer, bufferSize, MPI_INT, 1, tag, MPI_COMM_WORLD);
else
{
  MPI_Status s;
  MPI_Recv(buffer, bufferSize, MPI_INT, 0, tag, MPI_COMM_WORLD, &s);
}
MPI_Finalize();
```

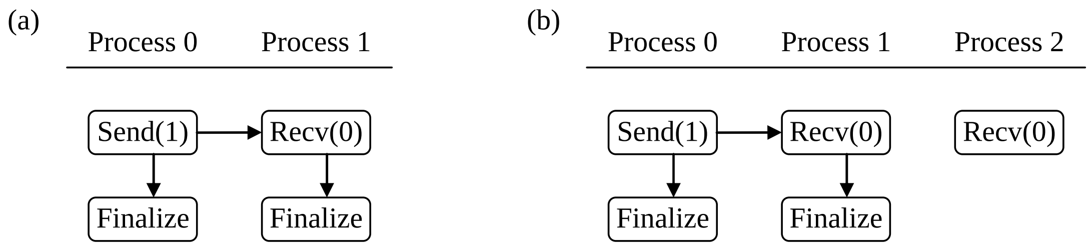


Figure 5.1: *(a) The expected execution of the code of Listing 5.8 with two processes $P0$ and $P1$; (b) the same code executing with three processes yields a deadlock due to an incorrect conditional clause.*

Figure 5.2a illustrates another deadlock situation caused by the use of inconsistent parameters when performing a barrier synchronization. Processes 0 and 1 perform a barrier on the *MPI_COMM_WORLD* communicator, while process 2 performs a barrier on another communicator.

Figure 5.2b displays a simple situation where a wildcard receive matches multiple sends despite the presence of a barrier synchronization. The use of non-blocking send and receives by processes 0 and 1 make the barrier call useless as it does not induce any dependencies between the various calls. In this case, moving the *MPI_Wait* call of process 1 before the call to *MPI_Barrier* is sufficient to prevent the race: by forcing the receive to complete before process 1 may enter the barrier, the wait call guarantees that the receive is matched before process 2 can send its message.

The *MPI_ANY_SOURCE* wildcard is designed to enable a single receive to match multiple sends. However, it is sometimes difficult to determine exactly the set of sends that is matched. The consequences of matching unexpected sends may vary. Different buffer sizes in the send and receive calls may be detected by the MPI library, which may then produce an error. In other cases, the execution continues but computation results may be incorrect. Finally, Figure 5.2c shows how a message race may lead to a deadlock. If the wildcard receive matches the send from process 2, both the synchronous send call and the second receive call from process 1 will be unable to complete.

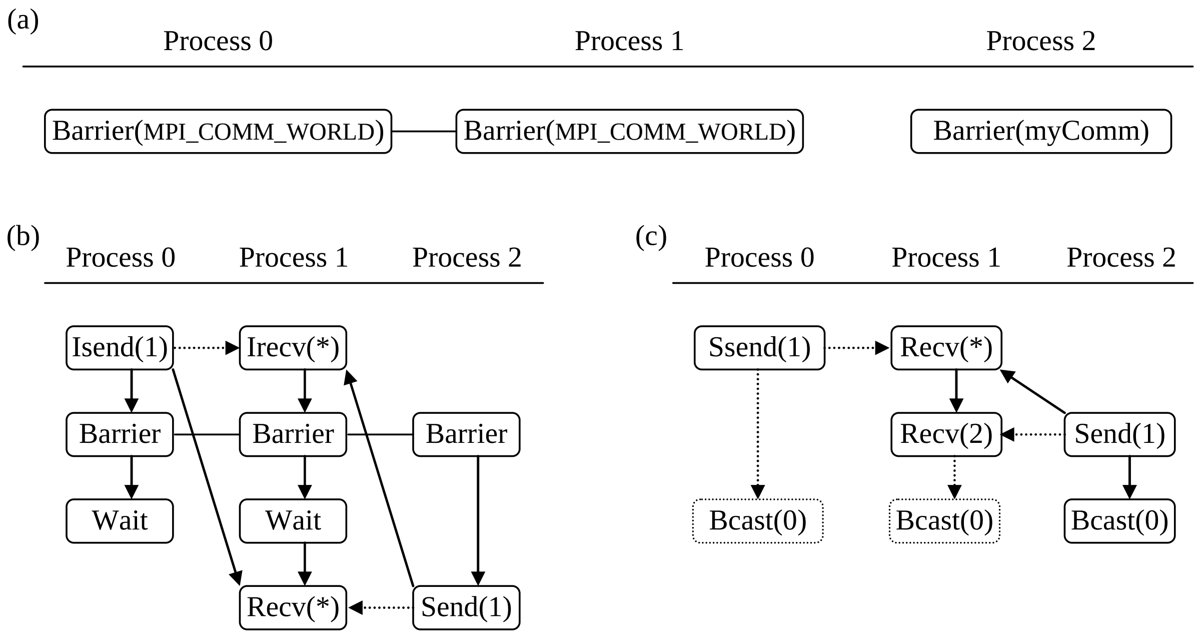


Figure 5.2: *(a) Mismatched parameters induce a deadlock: all members of a communicator must call a barrier call before any of them may proceed; (b) the use of non-blocking sends and receives makes the barrier call useless; (c) this program deadlocks only if the wildcard receive matches the send from process 2 rather than the one from process 0.*

5.4 Interactive testing of MPI applications

As we already mentioned in the previous chapter, there is a lack of tool providing a high-level view of the execution of parallel applications. This section applies the parallel application visualization concepts presented in Section 4.3 to MPI applications, and introduces a few MPI specific features. A graphical user interface displays the message-passing graph of the application and provides a high-level view of its communication patterns. Within the message-passing graph, we can hide or highlight MPI calls based on various criteria such as the originating process, the communicator on which the communication occurred, or the source code file or function that generated the call. We propose various types of high-level breakpoints to control the evolution of the participating processes. Execution scenarios that occur only rarely in actual executions can thereby be explicitly tested. Variants may be executed using an interactive replay functionality. A subset of potential conflicts over *MPI_ANY_SOURCE* receives may be detected automatically. Possible matches are then drawn on the message-passing graph, enabling the developer to decide which execution path must be followed by the application.

In addition to the related work presented in Section 4.1.2, several contributions have been developed specifically for detecting and replaying errors within MPI applications. Two contributions [14, 48] focus on record and replay techniques enabling reproducing a race once it has been detected. For instance, Retrospect [14] enables the deterministic replay of MPI applications, but the lack of control on the application execution may force the developer to run his application many times until an error is revealed. To our knowledge, ISP [121] is the only tool that explicitly tests different orderings of events within MPI applications. However, replaying an erroneous execution deterministically is only a first step in identifying a bug. The ability to visualize and to test slightly different executions often helps understanding the origin of an error and correcting it.

5.4.1 Architecture

The first component of the debugging functionality is an interception layer, implemented as a library that intercepts the MPI function calls performed by the application. The interception is carried out using the MPI Profiling Interface as described in Section 5.2.3. When the MPI initialization function *MPI_Init* is intercepted, every process opens a TCP connection to the debugger, a standalone Java program that receives and displays information about the current state of the application. The location of the debugger is specified using an environment variable.

Processes first identify themselves to the debugger by sending their rank and their process identifier. During the application execution, the interception layer then sends a notification to the debugger for every point-to-point and collective MPI function called. Notifications are also generated for the various *MPI_Wait* and *MPI_Test* functions, as well as for functions creating new communicators. With the exception of the message content, each notification contains a copy of all the parameters of the called function. These parameters may be MPI defined constants, such as *MPI_COMM_WORLD*, *MPI_INT* or *MPI_ANY_SOURCE*, whose value is specific to MPI implementations. The debugger receives a copy of these constants when the application starts, so as to be able to translate parameter values into human readable form when displaying information to the developer.

As one could see in the *MPI_Send* prototype in Listing 5.5, communicators and datatypes are respectively of types *MPI_Comm* and *MPI_Datatype*. The implementation of these types may differ for every MPI implementation. However, both types are represented by integers in the Fortran bindings of the MPI functions. We therefore use the C to Fortran conversion functions *MPI_Type_c2f* and *MPI_Comm_c2f* to convert all MPI datatypes and communicators into integers before their transmission to the debugger.

Notifications are sent to the debugger before calling the MPI function. Once it has sent a notification, a process suspends its execution and waits for an acknowledgment from the debugger. By withholding specific acknowledgments, the debugger may thus delay the execution of the associated processes while letting the rest of the application execute.

When wildcard receives are used, the debugger cannot automatically determine which source is actually matched. This information is provided separately by the interception layer via a *matched* notification. If the wildcard receive is blocking, the *matched* notification is sent immediately after the reception of the message by the receive function call. For non-blocking wildcard receives, the *matched* notification is sent when an *MPI_Wait* or *MPI_Test* call successfully queries the status of the non-blocking receive. In both cases, the rank of the matched source is read from the *MPI_Status* parameter of the querying call.

The debugger also needs to know about communicators in order to properly match send and receive calls. Each process therefore sends an additional *comm_created* notification after creating a new communicator. The notification includes the process rank in the new communicator, as well as the *MPI_COMM_WORLD* rank of process 0 of the new communicator. This second parameter identifies the “leader” process for the new communicator, and is for instance necessary for *MPI_Comm_split* calls, which subdivide a single communicator into a set of communicators. Since each new communicator contains a process of rank 0, the debugger will

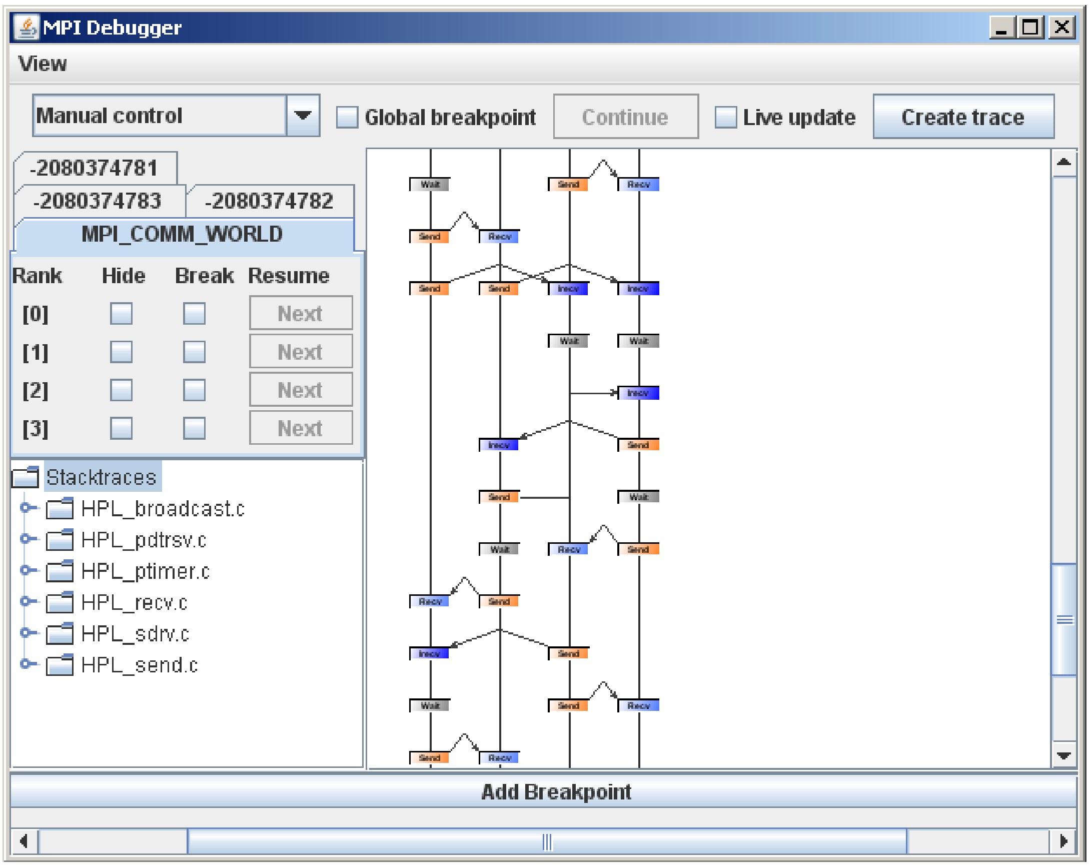


Figure 5.3: *Debugger window, displaying the zoomed out message-passing graph of HPL running on 4 processes. The left panels contain the list of processes and the stack trace tree.*

receive multiple notifications advertising the same rank. The "leader" process will be different however, and enables identifying the subsets of processes belonging to each one of the new communicators.

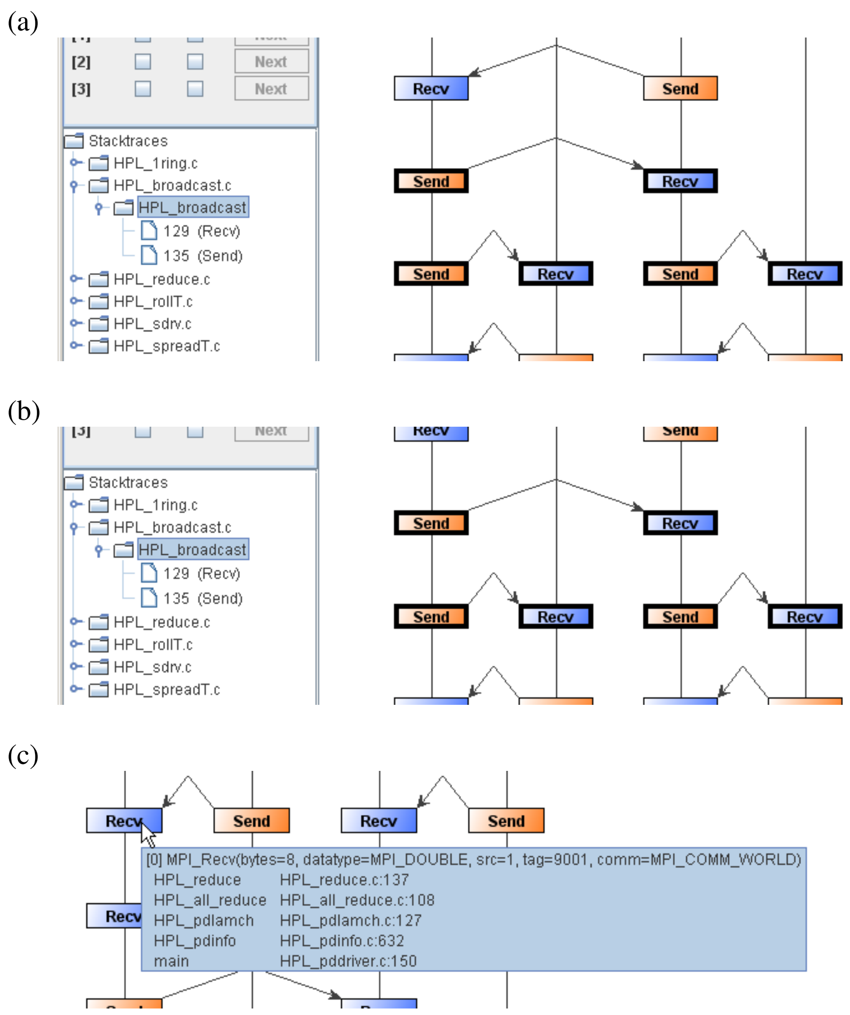
The user interface of the debugger consists of a single window that provides control elements to influence the application execution, and displays the current status of the application as a message-passing graph. The vertices of the graph represent the MPI calls performed by the application. Unlike most tracing tools that display time from left to right, our representation matches the one used within the MPI standard, where time flows from top to bottom. Vertices associated to notifications from a same process are therefore displayed one below the other, similarly to successive lines of code within a source file. Collective operations are grouped into a single vertex and are represented as a rectangle that spans all participating processes. The label and color of every vertex indicates the type of MPI operation executed, and tooltips display detailed information about the parameters of the call.

The debugger draws edges between successive vertices from a same process. It also draws edges of a different color between vertices associated to matching send and receive calls. For this purpose, the debugger maintains one *unmatched sends* and one *unmatched receives* queue. Upon receiving a notification for a send (resp. receive) call, the debugger looks for a matching receive (resp. send) call within the unmatched receives (resp. unmatched sends) queue. If none is found, the incoming notification is pushed at the end of the corresponding queue. When looking for matches, the queues are explored in a FIFO manner in order to respect the FIFO property of MPI communication channels. New vertices and edges are dynamically added to the graph as the debugger receives new notifications from the application. When the debugger receives a notification for a wildcard receive from a process p, it stops matching send calls with p as the destination until the reception of the corresponding *matched* notification. For non-blocking wildcard receives, graph updates are therefore delayed until p successfully queries the status of the non-blocking receive using a wait or test function.

Single-threaded processes cannot send more than one notification at a time to the debugger. The order in which the debugger receives notifications from a given process therefore matches the order of occurrence of events within that process, and the graph displays the temporal dependencies between these calls. In case of multithreaded processes where multiple threads may simultaneously call MPI functions (i.e. using *MPI_THREAD_MULTIPLE*), the message-passing graph no longer accurately represents the temporal dependencies between events. However, the interception layer makes sure that no two threads call MPI functions simultaneously, and that the order in which notifications are sent matches the ordering of MPI calls. The ordering of messages within communication channels is therefore known to the debugger, which may accurately display send-receive matches.

On Linux, the interception layer is able to determine the stack trace of every MPI call. A panel in the debugger window displays a tree containing the files, functions and line numbers from which the MPI functions were called. Nodes are dynamically added to the tree as new code locations are executed. Selecting a node of the tree then highlights all the associated vertices in the message-passing graph, illustrating how and when the selected file or function is used within the application. Figure 5.4 displays an example from the execution of the High Performance Linpack (HPL [25]) code.

Stack traces are also used for attaching a sequential debugger to application processes in order to inspect the actual application code. When the developer double-clicks the graph vertex of a suspended MPI call, the debugger opens a remote connection to the host running the calling process and attaches a user-specified sequential debugger to the calling application process. Thanks to the stack trace, it may then set a breakpoint to the source code line that immediately



```
[0] MPI_Recv(bytes=8, datatype=MPI_DOUBLE, src=1, tag=9001, comm=MPI_COMM_WORLD)
  HPL_reduce        HPL_reduce.c:137
  HPL_all_reduce    HPL_all_reduce.c:108
  HPL_pdlamch       HPL_pdlamch.c:127
  HPL_pdinfo        HPL_pdinfo.c:632
  main              HPL_pddriver.c:150
```

Figure 5.4: *Selecting a node in the stack trace tree highlights all calls performed from the associated source code location. Here, selecting HPL's (a) broadcast or (b) reduce functions instantaneously provides insight on the implementation of the broadcasting algorithm. (c) Stack trace information is also displayed within tooltips, along with the parameters of the MPI function call.*

follows the MPI function call. The debugger then sends the acknowledgment to the suspended process, which resumes and hits the breakpoint, enabling the developer to inspect the application code. Since notifications keep being sent to the debugger as the developer steps through the code of the attached process, the message-passing graph remains up-to-date.

In order to understand the communication patterns of an application, for instance when studying an application written by another developer, it may be useful to look at the whole message-passing graph. In other cases, one only needs to consider a subset of processes. We therefore provide the ability to zoom in and out of the graph in order to adapt its level of detail to the needs of the developer. A panel also displays the list of processes involved in the computation and enables hiding the graph vertices belonging to specific processes. When the application uses multiple communicators, the list of processes belonging to each one of them appears in additional tabs. When switching to a given communicator tab, the debugger highlights all communications carried out over the selected communicator (Figure 5.5). The developer may also choose to display a partial message-passing graph that includes only the vertices associated to MPI calls performing communications on the selected communicator.

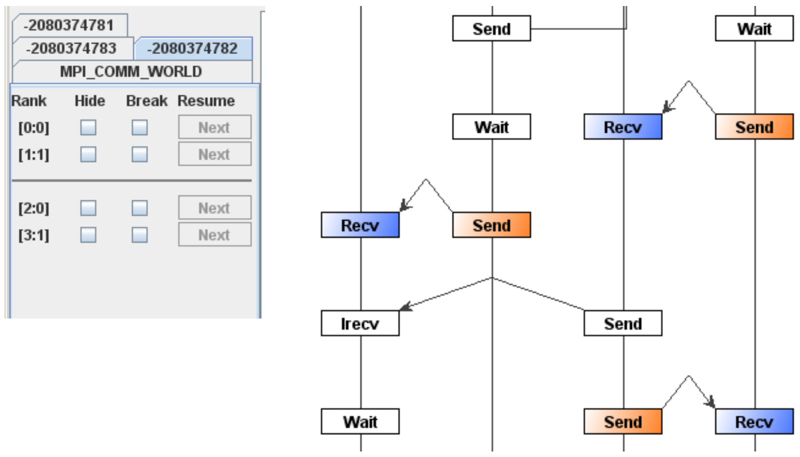


Figure 5.5: *Communications involving a particular communicator are highlighted by selecting the desired tab within the communicator panel.*

The interception library also intercepts calls from the MPI wrappers of the *autoserial* library. When the wrapper functions are used, the interception layer sends the full serialized object to the debugger, which may then display its content using a tree view similar to the one displayed in Figure 4.7. For objects to be understood by the debugger, the serialization is performed by the specialized textual serializer described in Section 2.7.4. The interception layer also provides functions for registering serializable objects representing the user-space state of the running application. The developer may retrieve and display these objects when a process is suspended by the debugger. The request is piggybacked on the acknowledgment for the pending notification of the selected process, causing the interception layer to send a copy of the registered objects.

5.4.2 Controlling the application execution

When the application starts, the debugger holds the first acknowledgment of every MPI process. A *Continue* button enables the developer to resume the application execution. If the developer does not activate any breakpoint or execution control mechanism, the debugger will immediately acknowledge all incoming notifications and let the application execute until completion.

A *global breakpoint* may be activated. It causes the debugger to withhold all acknowledgments, thereby suspending all processes. Clicking the *Continue* button then simultaneously acknowledges all pending notifications and resumes the execution of all processes up to the next MPI call. The global breakpoint allows quickly stepping through the execution of all processes at the message-passing level rather than at the instruction level, while maintaining the opportunity to take action on every notification.

Process breakpoints cause the debugger to systematically withhold the notifications sent by particular processes. A *Next* button aside of each process in the left panel of the debugger window must be pressed for the acknowledgment to be released. This feature may be used to arbitrarily delay specific processes in order to provoke message races. The developer may also explicitly test different execution orderings by breakpointing multiple processes and by resuming them in different orders. A finer control is provided via *conditional breakpoints*. They enable withholding acknowledgments for notifications matching one or several criteria such as the rank of the calling process, the type of MPI call, the message size or data type, or the destination rank for send calls. Moreover, the developer can specify a hit count to indicate how many times the breakpoint must be hit before it becomes active.

As illustrated in Section 5.3, the use of wildcard receives leads to non-determinism within the application execution. It may be difficult to identify potential conflicts, and manually controlling the application execution may be error prone. We therefore implemented a procedure that detects potential ordering variations on wildcard receives and allows the developer to choose the send call that matches a specific wildcard receive. When this mode is active, the debugger automatically acknowledges all notifications that are not associated to send calls. Upon receiving a notification for a send call, the debugger checks whether it already received a notification for a matching receive call. If not, it holds the send notification until it receives a matching receive notification. If the matching receive explicitly specifies the source of the expected receive, the debugger acknowledges the send notification, thereby resuming the process execution. If the matching receive is a wildcard receive, the debugger draws one large arrow between the graph vertices corresponding to the potentially matching send and the wildcard receive. Since the other processes keep running, more arrows may be added as the debugger receives other potentially matching send notifications. Clicking on one of the send vertices then acknowledges the associated notification. The resumed process then sends its message, which matches the wildcard receive under consideration. Section 5.4.3 illustrates the use of this functionality.

Since this scheme makes no assumption about whether calls should be blocking or not, it is able to reveal potential message races stemming from the buffering of messages within MPI calls. In Figure 5.6a, the debugger will acknowledge the broadcast notification from process 0. If process 0 buffers the broadcasted message, the debugger eventually receives a notification for the subsequent send call, which may match the first wildcard receive of process 1 if process 2 is delayed.

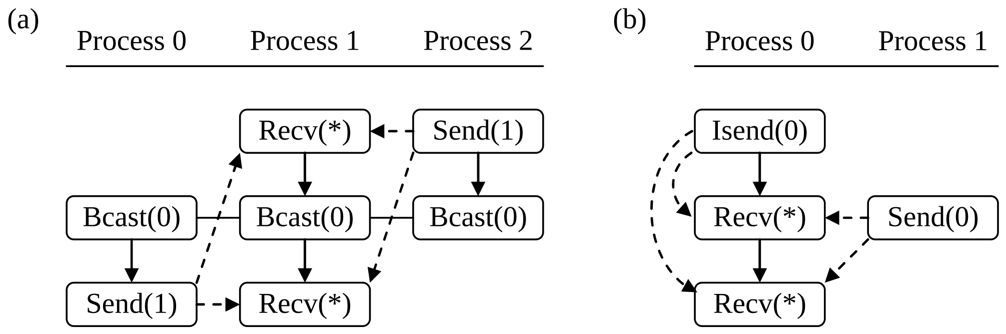


Figure 5.6: *(a) Send calls from both process 0 and process 2 may match both wildcard receives from process 1 if broadcast and send calls are buffered; (b) the debugger cannot receive the notification for the first receive from process 0 without previously acknowledging the non-blocking send from process 0.*

On the other hand, some executions involving non-blocking or buffered sends cannot be enforced. For instance, in Figure 5.6b the debugger cannot detect that a race could occur if the non-blocking send from process 0 is not acknowledged. Automatically holding all send calls may also suspend the execution of the debugger when non-blocking sends are used. This is the case in Figure 5.6b, where both processes are suspended by the debugger and none of them has any receive to match. Such cases must be manually resolved by clicking on one of the send vertices (in this example, on the Isend call from process 0) to acknowledge the associated notification and resume the execution. A solution to both problems has been proposed very recently in [118] and is presented in Section 5.5.6.

At any moment, the developer has the possibility of generating a breakpoint file. When the application restarts, loading the breakpoint file causes the debugger to set internal breakpoints that will reproduce the traced (and potentially incorrect) execution. During replay, the developer may set additional breakpoints to test execution variants. When the overhead of the debugger must be minimized or when the application cannot be run interactively, the interception layer is also able to generate a full standalone trace without being connected to the debugger. The trace file enables displaying the message-passing graph of the application for post-mortem analysis.

5.4.3 Debugging example

This section shows how the debugger can be used to induce and study a message race within a very simple parallel merge sort application. Listing 5.9 displays the MPI pseudo-code of the application. The code assumes that the number of processes p is a power of 2, and that the size n of the vector to be sorted is a multiple of p.

Process 0 initially sends one piece of the array to be sorted to every participating process, including itself. All processes then sort the received array part and call *MPI_Barrier*. The $p/2$ processes with the largest ranks send their array to one of the $p/2$ processes with the smallest ranks. The $p/2$ receiving processes merge together the received and local arrays. The $p/2$

Listing 5.9: *MPI pseudo-code for a parallel merge sort application.*

```
/* myRank:      process rank                */
/* n:           no of elements to be sorted  */
/* p:           no of processes (power of 2) */
/* procsInLoop: no of processes in the loop  */

MPI_Init()
/* ---------- Distribute array to nodes ---------- */
if (myRank == 0)
  for (i = 0 to p)
    MPI_Isend(array of size n/p to process i)
MPI_Recv(array of size n/p from process 0)
if (myRank == 0)
  MPI_Waitall(on MPI_Isend calls)

/* ------------- Sort received part ---------------*/
sort(array of size n/p)
MPI_Barrier(MPI_COMM_WORLD)

/* ---------------- Merge pieces ----------------- */

procsInLoop = p
do {
  if (myRank >= procsInLoop/2)
    MPI_Ssend(array of size (n/procsInLoop) to myRank%(procsInLoop/2))
  else {
    MPI_Recv(array of size (n/procsInLoop) from any source)
    /* Merge the received and local arrays */
  }
  procsInLoop = procsInLoop/2
}
while(myRank<procsInLoop && procsInLoop>1)

MPI_Finalize()
```

processes with the largest ranks then exit the loop, and the $p/2$ processes with the smallest ranks enter the next iteration. At every iteration, half of the remaining processes exit the loop. When all the pieces have been merged, the last process exits the loop and the application exits. Figure 5.7 shows the message-passing graph produced when running the application on four processes.

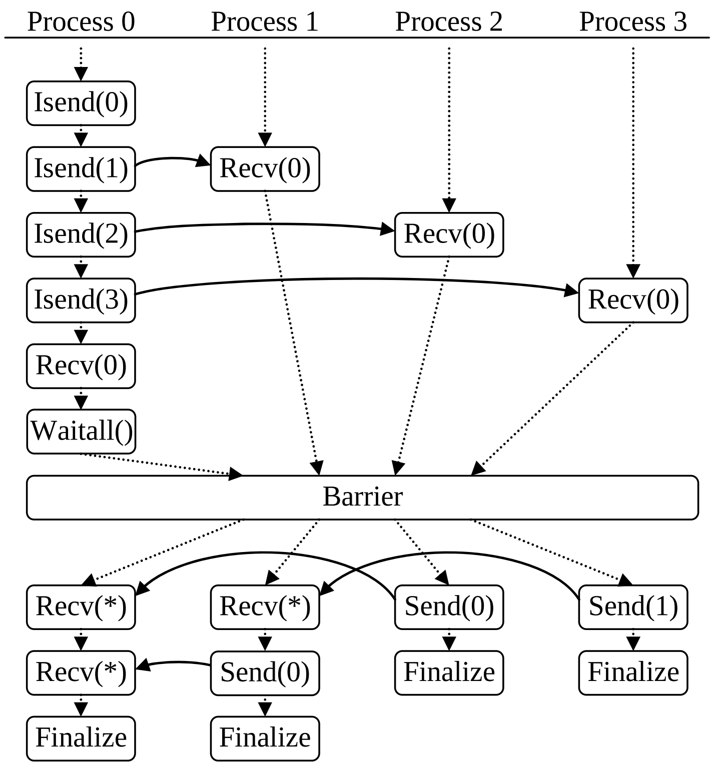


Figure 5.7: *Message-passing graph of a merge sort execution on four processes.*

The execution displayed in Figure 5.7 produces the expected result. However, it is not immediately visible from the code that at a single point in time, different processes may be executing different iterations of the *do while* loop. This can be deduced from the message-passing graph, which displays no causal dependencies between the send calls of processes 1 and 2 to process 0. If both calls use the same tag and if the destination only performs wildcard receives as illustrated in Figure 5.7, then a message race may occur.

A first way to test such an execution is to activate the process breakpoint of process 2 and leave the other processes unhindered. Process 3 sends its message to process 1, which then sends its message to process 0. Since process 0 in the first MPI call after the barrier, it is expecting a buffer containing $n/4$ elements. Process 1 however is already in the second call after the barrier, and sends a sorted subarray containing $n/2$ elements. Resuming process 2 then

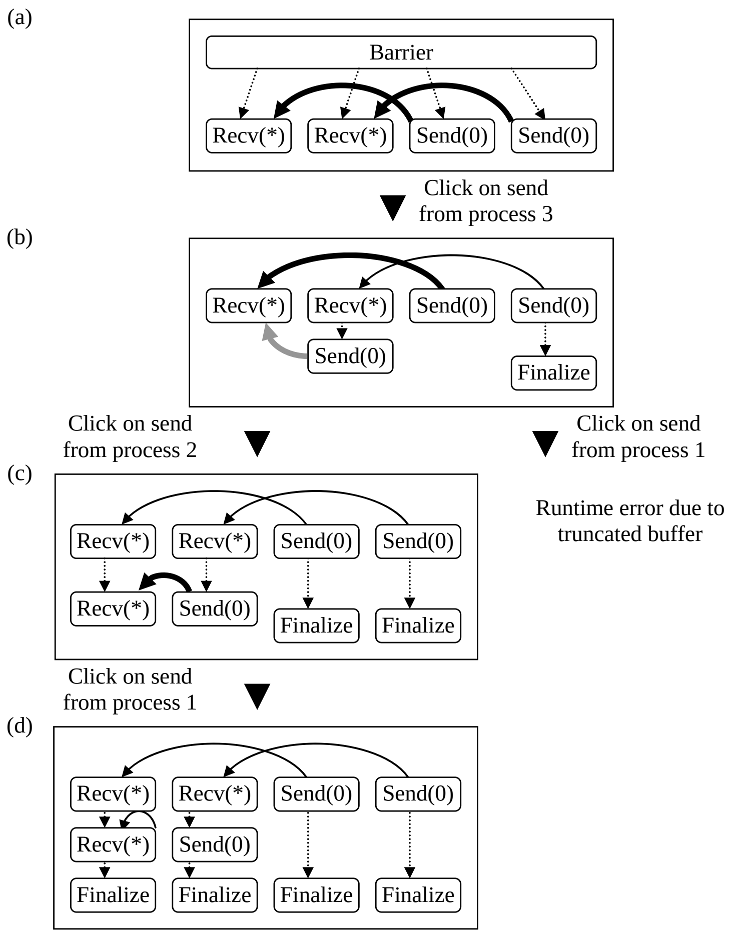


Figure 5.8: *Possible sequences of steps produced by the wildcard receive detector. Bold edges indicate possible matches detected by the debugger. The send call is acknowledged by clicking on the associated vertex within the message-passing graph. The gray edge indicates that the size of the messages do not match.*

causes the receive call to truncate the incoming message and produces an error. The same result can be obtained by setting a single conditional breakpoint on *MPI_Send* calls from process 2.

Figure 5.8 depicts how the race is revealed by the automated wildcard receive detection. Since no wildcard receives are performed before the barrier synchronization, the debugger automatically acknowledges all notifications. After the barrier, the sends from processes 2 and 3 both match a wildcard receive. The debugger therefore withholds the associated acknowledgments and displays the potential matches that can be chosen by the developer (Figure 5.8a). If the developer clicks on the send from process 3, the debugger acknowledges the associated notification. This resumes process 3, which then sends its message to process 1, which resumes its execution following the message reception. The debugger then receives the next notification from process 1 and detects that the send calls from both process 1 and process 2 may potentially match the same receive call from process 0. It draws an edge of a different color to indicate that message sizes do not match between process 1 and process 0 (Figure 5.8b). From that point, clicking on the send event of process 2 enforces a correct execution, while clicking on the send event of process 1 causes the receiving call to truncate the message and the MPI library to raise a runtime error[2].

The graph may also be used to identify potential performance improvements. In our small example, we first see that since processes use non-wildcard blocking receives to collect their piece of the array, the barrier call is useless. We also see that process 0 sends the pieces of the array using non-blocking send calls to every process including itself. Tooltips indicate that all messages are of the same size, and the stack trace of each call indicates that all non-blocking sends are called from the same line of source code, i.e. within a loop. The initial distribution could therefore likely be carried out through a single *MPI_Scatter* call.

5.4.4 Impact on applications

Running an application under the control of a scheduler may alter the ordering of events within the application. Races appearing in regular executions may thus disappear when running under control of the debugger. Nevertheless, with the exception of the issues about wildcard receives discussed above, most races stemming from nondeterministic orderings of MPI calls can be explicitly induced by appropriately setting breakpoints within the application.

A second concern is the application slowdown. Since it must process all the notifications sent by the processes, the debugger becomes a bottleneck when the rate of incoming notifications increases. In order to evaluate that impact, we performed measurements on the Pleiades cluster at EPFL, which consists of 132 single-processor nodes connected with a Fast Ethernet switch. We ran the High Performance Linpack [25] benchmark on four nodes, with one process per node and a 100×100 matrix decomposed into 10×10 blocks. This run called 664 MPI communication functions during its running time of 0.03 seconds, leading to a call rate of 22'000 calls per second. Once connected to the debugger, the same application ran in 30 seconds, or about 1000 times slower. For this test, the debugger was therefore able to process and display about 22 notifications per second. While such a display rate is sufficient to manually step through the application execution, the developer often wants the application to execute quickly up to the point where he wants to start his analysis.

The major portion of the slowdown is due to the display of the events to the application

[2]This is at least the behavior of MPICH2. Other MPI implementations may behave differently.

developer. If we run the debugger without layouting and displaying the graph, the running time falls to 0.16 seconds, reducing the slowdown compared to the original application running time to a factor of 5. Figure 5.9 presents the application slowdown factor as a function of the average number of MPI calls per second performed by all processes during the execution. It displays results for HPL running on 4, 8 and 16 nodes with one process per node for various matrix and block sizes (from 2000 to 8000 and from 25 to 500 respectively). For a given number of nodes, the slowdown can be approximated fairly well using a linear function. The slope becomes less steep as the number of nodes increases, due to the fact that the debugger uses one thread per MPI process to receive and acknowledge the notifications. Since the notifications are well balanced between the processes, the multithreading improves the overlapping of processing and communication within the debugger.

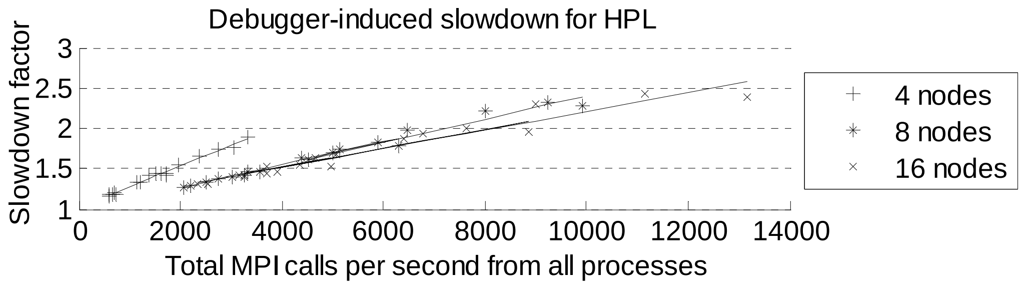


Figure 5.9: *Application slowdown when connected to the debugger, when the display of the graph is disabled. The call rate of MPI functions is computed when the application is not connected to the debugger. Results for 4, 8 and 16 nodes are well approximated with a linear function.*

These results show that high notification rates may occur. It is therefore crucial that we optimize the debugger's layout and display code in order to achieve better performance. The performance can currently be slightly improved by disabling the live updating of the message-passing graph, which is then refreshed at once when a breakpoint is hit or when the developer explicitly requests an update.

5.4.5 Improving scalability

Scalability issues are similar to the ones encountered in Section 4.3.5 while visualizing the execution of DPS applications. They lie on one hand on the performance impact of the debugger on the running application, and on the other hand on the overwhelming amount of information displayed when large graphs are produced. Improving the performance of the current code is therefore insufficient, and future work should focus on (1) reducing the amount of data transmitted between the application and the debugger, (2) improve the display of the message-passing graph to facilitate its analysis and (3) automate the detection and reporting of meaningful events such as deadlocks or application crashes.

The interception layer currently implements the *MPI_Pcontrol* function to enable and disable the sending of notifications to the debugger. While this solution requires changing the application code, it has two major advantages:

- In addition to increasing the running time overhead, high notification rates lead to large graphs that are difficult to analyze. Disabling the transfer of notifications between the application and the debugger therefore reduces the amount of data transmitted over the network and limits the number of graph updates. Fewer updates not only limit the time consumed for layouting and displaying the message-passing graph, but also reduce the size of the message-passing graph displayed by the debugger.

- By controlling the display of the information, a developer is able to display a partial message-passing graph that only displays information that is relevant to his analysis.

The use of the *MPI_Pcontrol* function could be extended in order to provide a finer control over the types of MPI calls that should be transferred to the debugger. Regarding the reduction of application-debugger communications, we could use asynchronous notifications while using an automated running mode such as the one described in Section 5.4.2.

5.4.6 Future extensions

One interesting extension would be to provide the ability to label the message-passing graph, e.g. using expressions of the form *MPI_Pcontrol(ENABLE, "Iteration 1").* Figure 5.10 displays an illustration of the potential results on the graph of the merge sort application displayed in Figure 5.7. The ability to search for events or to collapse parts of the graph, e.g. between two barrier synchronizations or collective communications, would also facilitate the visualization of large graphs.

It would also be very interesting to integrate checkpoint/restart capabilities [14, 26, 94] into the message-passing graph based debugger. Combined with the provided control on the application execution, this feature would enable interactively testing multiple execution scenarios without requiring reexecuting the application from the beginning.

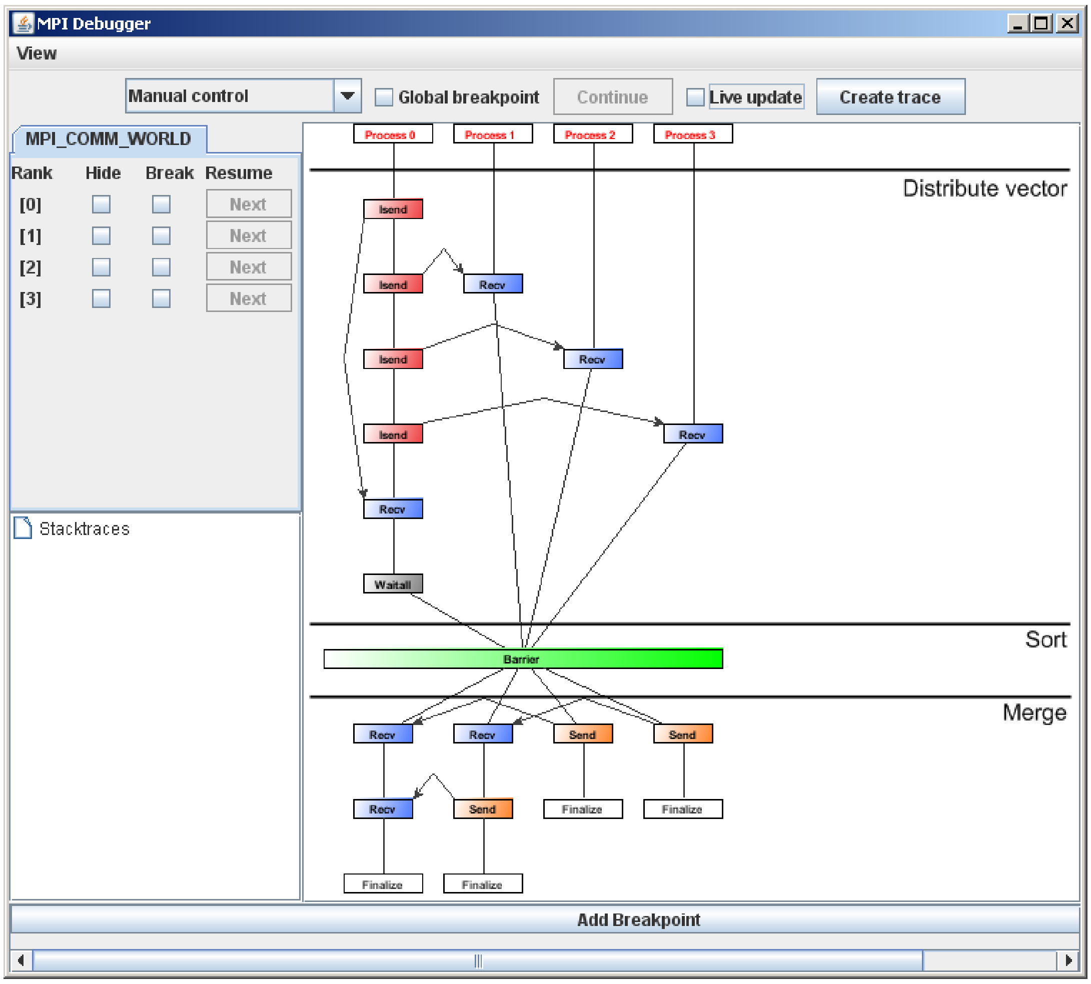


Figure 5.10: *User interface sketch of adding annotations to the message-passing graph.*

5.5 Automated error detection and testing of MPI applications

We now explore various ways for automatically testing MPI applications. The results in this section are somewhat preliminary, and many of them have been superseded by ISP [121], a tool developed very recently by Ganesh Gopalakrishnan's team at the University of Utah. We nevertheless present our results in order to illustrate the difficulties encountered while testing MPI applications, and to show how the techniques presented in Chapter 4 can be applied to parallel programming models other than DPS.

We first present a scheme for detecting deadlocks in MPI application in Section 5.5.1. In Section 5.5.2, we study the suitability of partial-order execution graphs (POEG) to represent the execution of MPI applications. We then show how they can be used to detect message races within applications producing a fixed set of messages (Section 5.5.3) and to automatically identify potential performance improvements (Section 5.5.4). Section 5.5.5 then sketches a dynamic testing method based on the messages-passing state graph construction described in the previous chapter (Section 4.5). Section 5.5.6 then briefly presents the key ideas behind ISP and lists the similarities and differences with our work.

5.5.1 Detecting deadlocks in MPI programs

Two tools are explicitly targeted at the detection of deadlocks in distributed memory MPI programs. MPIDD [45] uses the MPI profiling interface to wrap calls and send relevant information to an external detector program, which detects deadlocks without relying on a timeout. However, false deadlocks may be reported if notifications are delayed. Processes therefore wait for an acknowledgment from the detector before continuing their execution. MPI-CHECK [73] is a decentralized tool targeting Fortran programs. The detection is performed via handshaking code inserted before blocking calls, and deadlocks are detected if the handshake does not complete within a user-defined timeout period. The timeouts are required due to the lack of a central coordinator with a complete view of the system. Umpire [122] looks for both resource leaks and deadlocks. Processes communicate through shared memory with a centralized manager. Like in MPIDD, processes are suspended until they receive an acknowledgment from the manager.

We propose a fully asynchronous scheme relying on a central coordinator. The detection can therefore be performed either online during application execution, or offline based on execution traces. Our implementation supports C, C++ and Fortran MPI programs, and requires no modification of the application source code. The implementation is portable and has been tested on both Linux and Windows.

Our technique detects both actual and *potential* deadlocks. Following the definition from [73], potential deadlocks describe deadlocks whose occurrence depends on the underlying MPI implementation or on the size of messages being transfered, i.e. depending on whether *MPI_Send*, fan-in and fan-out collective functions are buffered or not.

Our detector uses the same interception library as the visual debugger described in the previous section. It therefore sends a notification to the detector for every MPI function call, for the creation of new communicators, and uses *matched* notifications to indicate the source of successfully matched wildcard receives Section 5.4.1. In *online* mode, the deadlock detection code runs directly within the interception library, within a separate thread on the process of rank 0. All communications between the application processes and the detector are carried out using MPI calls, and therefore take advantage of the potentially available fast network interconnects. The interception library uses its own private communicator to properly segregate its own communications from those of the application. In order to minimize the overhead caused by having a centralized detector, the interception library sends notifications to the detector asynchronously and does not wait for an acknowledgment from the detector.

Using a distinct thread within one of the existing MPI processes avoids the use of an external server process, and enables running a hooked application identically to the regular application. The drawbacks of the use of a distinct thread are that the detector may compete with the application for CPU cycles, and that it requires that the underlying MPI library supports the *MPI_THREAD_MULTIPLE* level of thread-safety.

The running time overhead and the requirement for a library with multithreading support can be overcome by using the detector in *offline* mode. The interception layer then produces a trace, i.e. each process produces one file containing the list of MPI calls and their parameters. The trace may then be loaded and analyzed by a separate program running the detector component. This approach minimizes the impact of the interception library on the application running time, and is therefore useful for large applications, when the cluster utilization is expensive or when it should be minimized. The offline mode can also be convenient when parallel jobs are managed using a batch system, which does not allow users to decide when their application will run.

The detector always processes the events from a given process in the order in which they have been produced by the process. In online mode, this ordering property follows from the FIFO link property of MPI communication channels, which guarantees that notifications of MPI calls are received in the order in which they have been sent. In offline mode, the ordering property follows from the fact that each process trace is written sequentially.

Point-to-point and collective communications are handled separately within the deadlock detector:

Point-to-point communications. The deadlock detector maintains a reception queue and a send queue for every process of the parallel application. Upon reception of a send event that specifies a process p as its destination, the detector tries to match a receive event in the reception queue associated to process p. Similarly, upon reception of a non-wildcard receive event from a process p, the detector tries to find a matching event in the send queue associated to process p. If a match is found, the matching receive or send event is removed from the queue; else, the event received by the detector is appended to the appropriate queue. The handling of wildcard receives is described below.

Collective communications. The deadlock detector maintains a list of incomplete collective operations, i.e. collective operations that miss events from some of the participating processes. Multiple collectives may be simultaneously active when the application uses multiple communicators. Upon reception of a collective event from a process p, the detector looks within its list for a matching collective operation that expects the participation of p. If a match is found, p is added to the list of participating processes; else a new collective operation is appended to the list. If p is the only missing process, the collective is complete and is removed from the list.

The deadlock detector distinguishes *blocking* and *non-blocking* events. Non-blocking events are the notifications associated to instantaneous send and receive variants, such as *MPI_Isend*, *MPI_Issend* and *MPI_Irecv*. All other point-to-point events, as well as collective and wait events are considered to be blocking. This classification interprets the MPI standard strictly and ignores the fact that an MPI implementation may buffer small messages internally for performance reasons. Since the buffering strategy is unspecified, parallel codes that rely on that behavior may suddenly deadlock when the size of messages increases or when another MPI library is used. We use the distinction between blocking and non-blocking events to keep track of whether a process is *running* or *blocked*. A process is *blocked* when it contains a blocking event in one of the send, receive or incomplete collective operation queues. Otherwise, it is *running*. Incoming events of processes marked as *blocked* are queued by the detector. These events are only processed once their source process becomes *running* again. This buffering of incoming notifications avoids accumulating dependencies from one process on all others and solves the erroneous detection of inexistent deadlocks reported by [45].

When a process p becomes *blocked*, the detector adds dependencies from p to the processes that are expected to produce matching events that would resume p: a blocking send event causes a dependency on the destination process, while a non-wildcard blocking receive event causes a dependency on the specified source process; collective events cause dependencies to every other process member of the specified communicator. Wait and Waitall events cause a dependency to be added for every associated request that is still unmatched. Pairs of dependencies connecting two processes in opposite directions cancel each other if their source events can be matched. Remaining cycles reveal a deadlock.

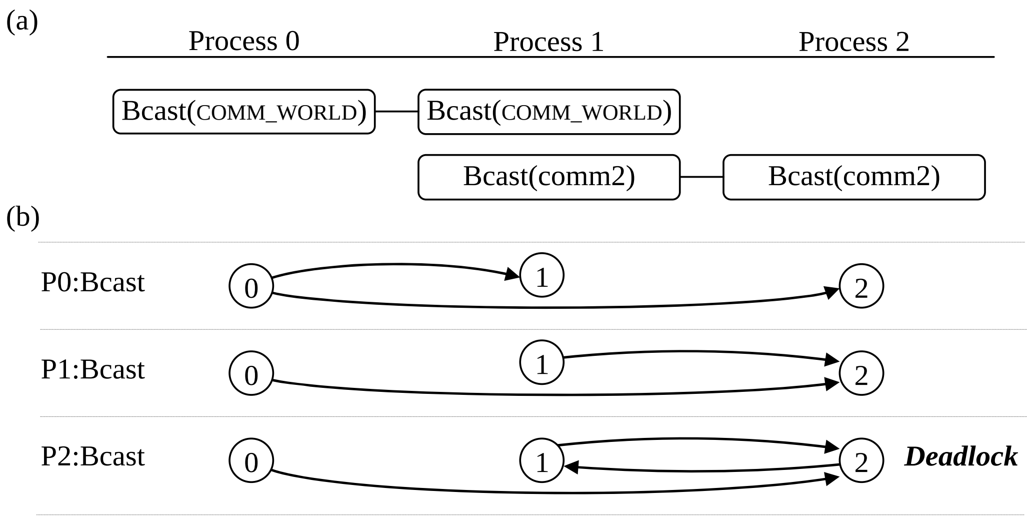


Figure 5.11: *(a) Process 2 fails to participate in the first collective broadcast; (b) upon reception of the broadcast event from process 0, the deadlock detector adds dependencies to other processes of* MPI_COMM_WORLD*, i.e. processes 1 and 2. The dependency to process 1 is removed upon reception of the matching broadcast event from process 1. Process 2 calling* MPI_Bcast *with incompatible parameters (here a different communicator) introduces a loop within the dependency graph, which reveals a deadlock.*

Figure 5.11 displays an example where one process fails to participate in a collective communication. The dependency of process 0 on process 1 caused by the broadcast call from process 0 is canceled when process 1 calls a broadcast function with matching parameters. A cycle is however detected when process 2 fails to call *MPI_Bcast* with the correct parameters. Figure 5.12 illustrates an incorrect reliance on buffered *MPI_Send* calls. Figure 5.13 shows two examples with non-blocking events.

Since the detector cannot identify the matching source of a wildcard receive, it matches them slightly differently. Upon receiving a wildcard receive, the detector does not add any dependency on other processes. If the receive is blocking and if the detector finds a match among pending send calls, the detector has the guarantee that it will eventually receive a *matched* notification from the receiving process. It therefore delays the match until receiving the *matched* notification. A deadlock is reported if and only if all processes of the communicator either called a wildcard receive, or transitively depend on a wildcard receive. For example, a process calling a barrier on *MPI_COMM_WORLD* will transitively depend on any process that called a wildcard receive.

Non-blocking wildcard receives are however slightly more tricky. The detector optimistically matches non-blocking wildcard receive events with the first matching send event. However, this may cause a discrepancy between the match performed by the detector and the actual

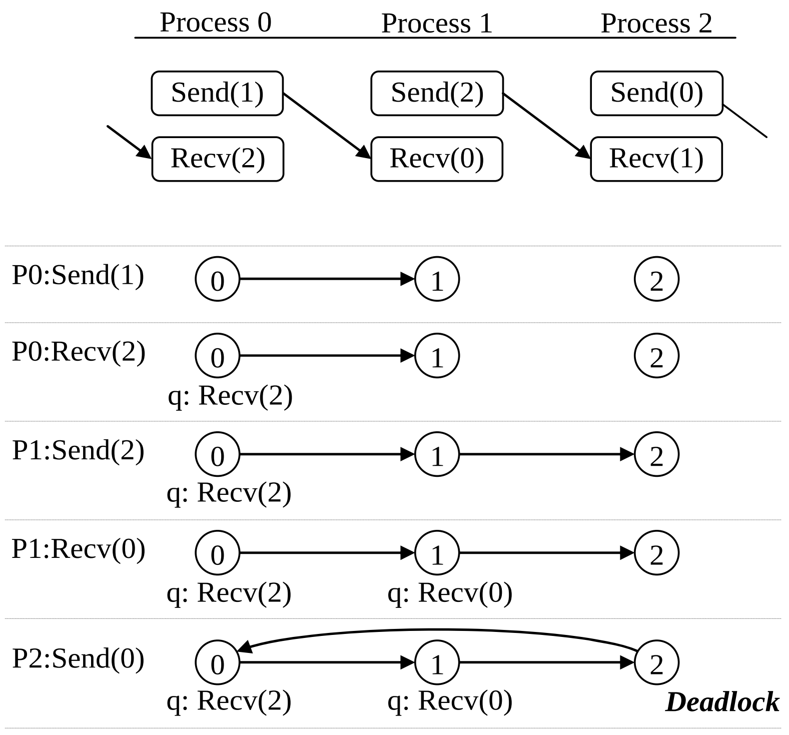


Figure 5.12: *Cycle of* MPI_Send – MPI_Recv *calls. The deadlock may not occur in practice if send calls are buffered. Since send events are designated as* blocking, *the detector queues subsequent receive events are queued (e.g.* q: Recv(2)*) until the previous blocking call is matched; failure to do so would enable the send from process 2 to match the receive from process 0.*

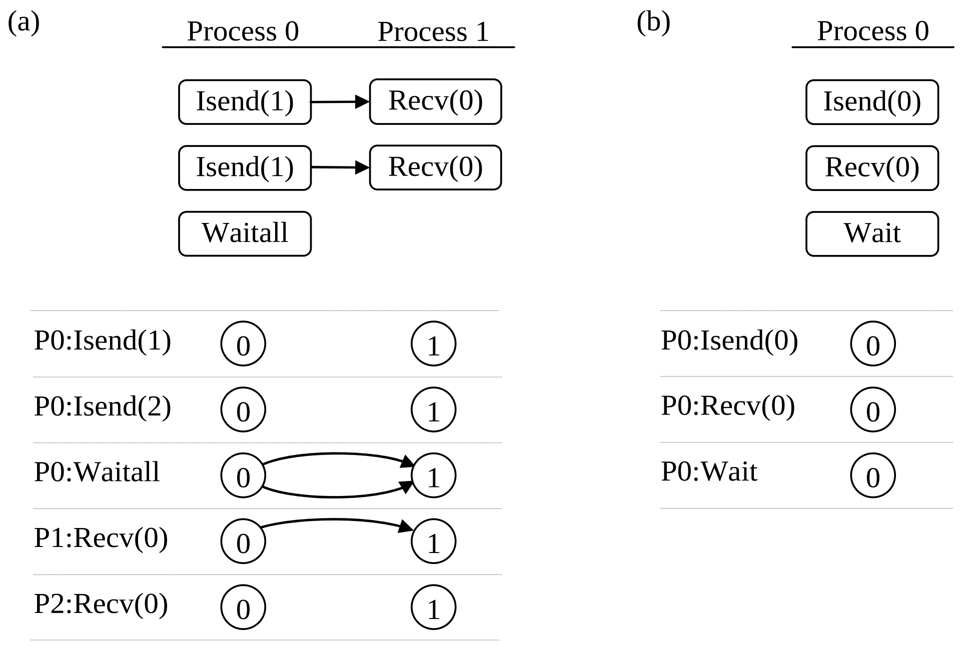


Figure 5.13: *(a) Upon reception of the wait all event, the deadlock detector adds dependencies to the destination of the unmatched non-blocking send events; (b) the* MPI_Recv *is correctly called before the* MPI_Wait, *thereby avoiding a self-deadlock.*

match within the application. A first consequence is that the detector may report errors that could occur if the wildcard receive matches a different source (Figure 5.14a). More importantly, some deadlocks may be missed as shown in Figure 5.14b.

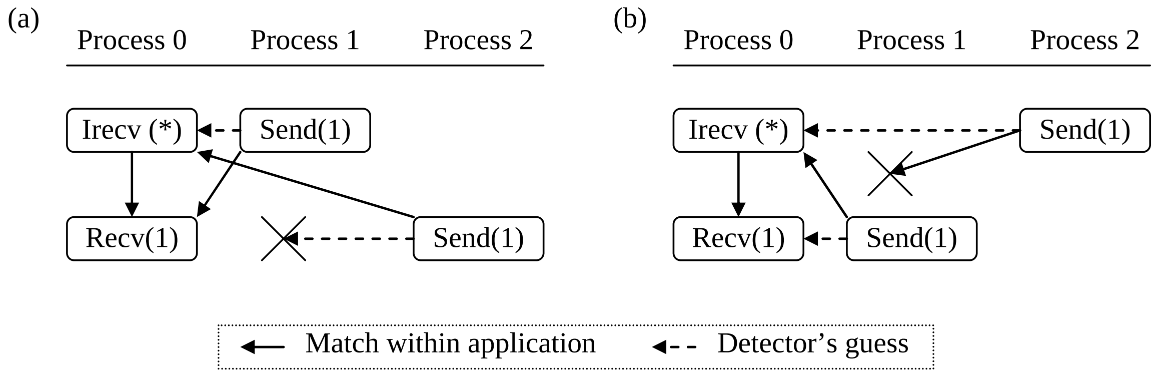


Figure 5.14: *(a) If the detector erroneously matches the wildcard receive with the send from process 1, it will report a potential deadlock; (b) if it erroneously matches the send from process 2 however, it will fail to detect the deadlock.*

Non-blocking wildcard receives are hard to analyze statically. For instance, [73] explicitly indicates that MPI-CHECK does not support wildcards in non-blocking receives, and [122] does not provide sufficient information to indicate that they do not suffer from that problem as well. The technique used by ISP also provides a solution to this problem [118], and is discussed in Section 5.5.6.

5.5.2 Partial-order execution graphs for wildcard-free MPI applications

Section 4.4 presented a partial-order execution graph decomposition technique for DPS applications producing a fixed set of messages. The decomposition can however be applied to any application that can be modeled using a POEG. In [107], Siegel and Avrunin describe an event model for wildcard-free MPI applications. We adapt that model to derive the POEG of a simple wildcard-free MPI application.

Listing 5.10 and Figure 5.15 respectively show the MPI pseudo-code and the corresponding POEG for an iterative neighborhood-dependent computation similar to the one discussed in Section 4.2. The model described in [107] distinguishes send and receive events. Each *MPI_Sendrecv* call is represented using a pair of events: a *send* event reads the send buffer parameter of the function, while the *recv* event writes into the receive buffer. The edges of the POEG connect send events to their matching receive, and events from blocking MPI calls to the events of the subsequent call performed by the program. A send from process i to process

Listing 5.10: *MPI pseudo-code for the border-exchange phase of a neighborhood-dependent iterative computation.*

```
/* np: Number of processes, rank: Process rank */
for (i = 0; i < maxIter; ++i) {
  if(rank == 0)
    MPI_Send(lower border to proc. 1);
  else if (rank < np-1)
    MPI_Sendrecv(lower border to proc. rank+1,
                 lower border from proc. rank-1);
  else /* rank == np-1*/
    MPI_Recv(lower border of proc. np-2);

  if(rank==0)
    MPI_Recv(top border of proc. 1);
  else if (rank < np-1)
    MPI_Sendrecv(top border to proc. rank-1,
                 top border from proc. rank+1);
  else
    MPI_Send(top border to proc. np-2);

  /* update subdomain */
}
```

j at iteration k is denoted as $s^k_{i,j}$, while $r^k_{i,j}$ denotes the matching receive event.

The ordering of events on individual processes is fully determined, except for pairs of send and receive events caused by *MPI_Sendrecv* calls. Nevertheless, the number of event orderings grows exponentially, with 60, 6268 and $6.5 \cdot 10^5$ orderings for 1, 2 and 3 iterations on 3 processes respectively.

It is however possible to decompose the POEG according to the principles described in Section 4.4. We first separate the contributions of the individual processes, and then identify

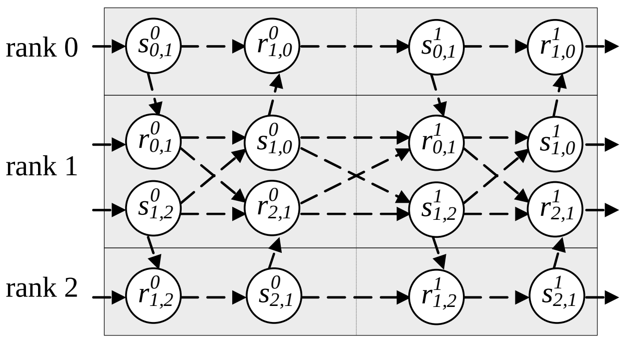


Figure 5.15: *Partial-order execution graph for two iterations of an MPI implementation of the neighborhood-exchange computation of Figure 5.10 when running on three processes.*

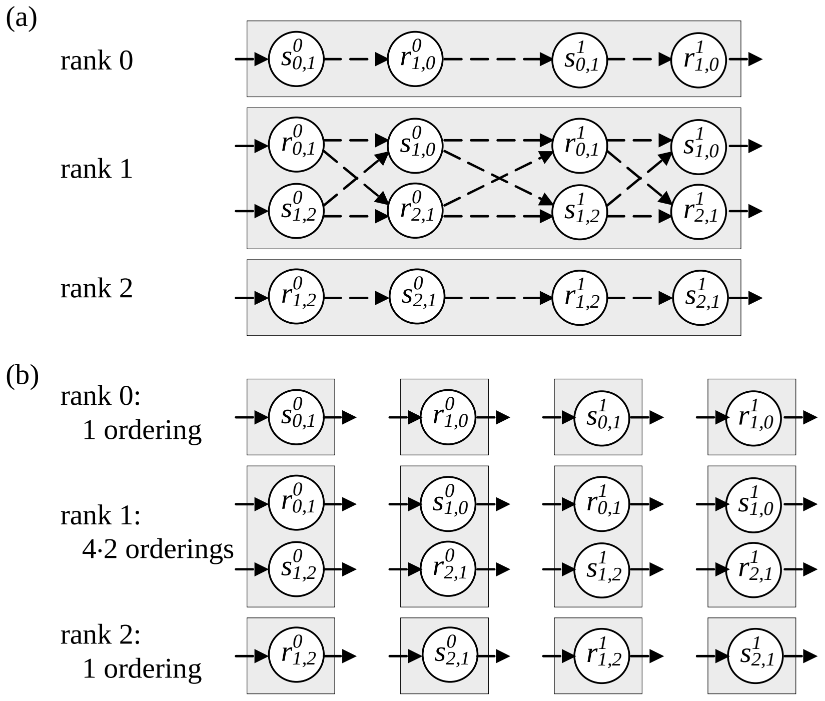


Figure 5.16: *(a) process partitioning and (b) subgroup partitioning and resulting number of orderings. If the send and receive buffers of* MPI_Sendrecv *calls do not overlap, an edge can be added between the associated send and receive events to further reduce.*

individual subgroups of events within each process. Once the decomposition is performed (Figure 5.16), the number of orderings grows linearly with the number of processes and the number of iterations.

5.5.3 Generalized partial-order execution graphs

We now study a generalization of POEGs to wildcard receives and buffered calls. Unlike the deadlock detector which interpreted the MPI specification in a strict sense, detecting message races requires taking into account the fact that calls may be buffered. We derive that information by building a Partial Order Execution Graph of the application that expresses the dependencies between the MPI calls produced by the application. Unlike the POEGs of DPS applications, an edge of which represents Lamport's *happened-before* relationship [64], edges of such graphs have slightly different meanings depending on the MPI calls they connect:

- If two MPI calls c_1 and c_2 are executed by the same process, an edge from c_1 to c_2 indicates that c_2 cannot occur before c_1 completes.
- If two MPI calls d_1 and d_2 are associated to events from different processes, an edge from

d_1 to d_2 indicates that d_2 cannot complete before d_1 occurs.

As the detector receives notifications of the MPI calls performed by the application, it builds a graph to define the dependencies between these events. Blocking calls are thus *MPI_Ssend*, *MPI_Recv*, *MPI_Wait*, and the receive part of *MPI_Sendrecv*. Regarding collective communications, synchronizing collective calls of all processes, fan-in calls of the root process, and fan-out calls of all but the root process are blocking. The graph building process starts from a single root node, which is associated to the *MPI_Init* call of all processes. We keep track of the *last blocking node* of each process. Since it must complete before any subsequent call may occur, the last blocking node then becomes a predecessor of all subsequent MPI calls produced by the same process. Additionally, and except for *MPI_Isend* which may be buffered, edges are added between non-blocking point-to-point calls and their matching wait. Figure 5.17 displays the resulting snippets for a few common constructions.

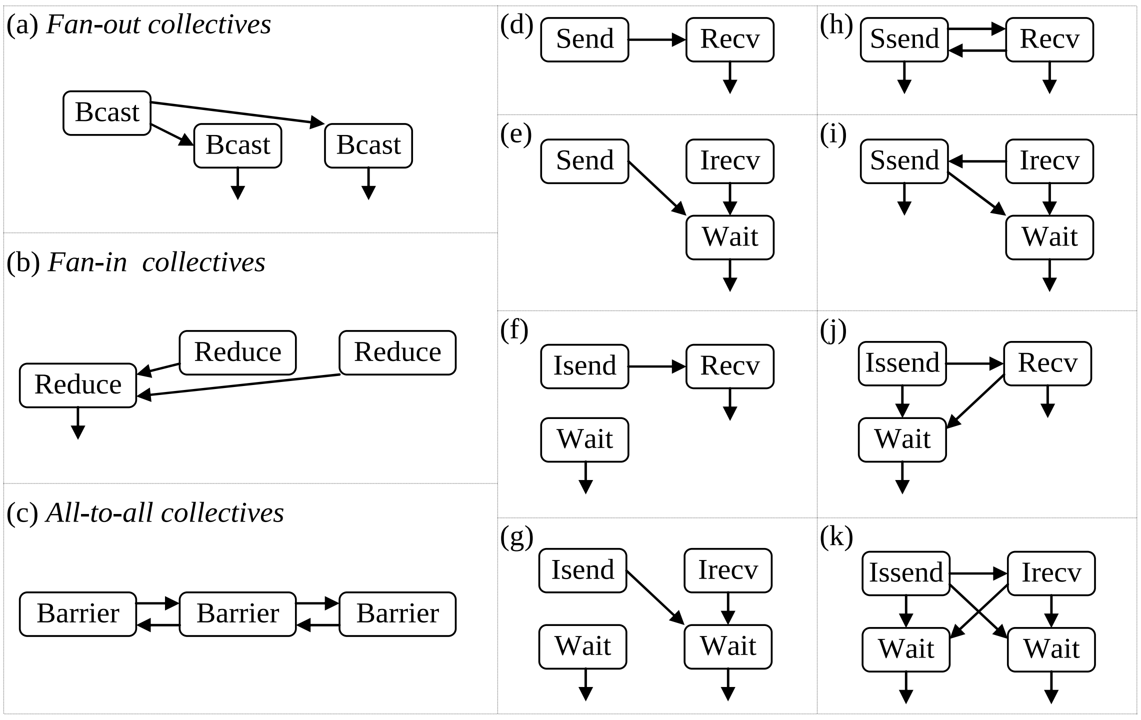


Figure 5.17: *POEG dependencies for various common MPI constructions. Nodes with an outgoing edge become* last blocking node *of their process when they occur. (a–b) The root of fan-out, resp. fan-in collectives is the predecessor, resp. successor of the other events; (c) events of synchronizing collectives are successors of each other; (d–k) dependencies for various combinations of send and receive calls: blocking receives and the wait of non-blocking receives are successors of their matching send, while only synchronous sends become predecessors of subsequent calls.*

Collective operations are handled as follows. While synchronizing collective communications may be represented by a single node, the representation of fan-in and fan-out collective calls must take into account the fact that MPI implementations are allowed to buffer the arguments of an operation on some processes for performance reasons. We therefore maintain one node for every event, and add edges such that the event from the root of fan-out collective communications becomes a predecessor of all other events, and the root of the fan-in collective communication becomes a successor of the other events. Synchronizing (i.e. all-to-all) collective events are all predecessors of each other (Figure 5.17c).

Given such a POEG, the set of sends matching a wildcard receive is the set of sends that are not successors of that receive. Unfortunately, this definition holds only for the first wildcard receive of each process: while one can still compute the set of matches of subsequent wildcard receives, these sets may include send calls potentially matched by earlier receives. Moreover, the POEG does not take into account the FIFO property of MPI communications, which may also impact the set of possible matches of subsequent wildcard receives. Indeed, adding links between subsequent sends from the same source to the same destination transitively introduces unwanted dependencies: for instance, in a *MPI_Isend – MPI_Ssend – MPI_Barrier* sequence, the barrier would become a successor of the non-blocking send. Figure 5.18 shows how the set of matching sends of subsequent wildcard receives depend on the chosen match of previous receives.

These results imply that each application is in fact described by a family of POEGs. Testing an application for message races would therefore require (1) generating all possible POEGs, (2) decomposing each POEG along process boundaries, and (3) testing each process for the possible ordering of events. Being a static structure, the POEG also assumes that the set of events produced by the application is unrelated to the order in which messages are actually received.

We illustrate this procedure for the merge sort example application illustrated in Section 5.4.3. Figure 5.19a displays the two possible POEGs when the application runs on four nodes. Each POEG may then be decomposed along process boundaries. The decomposition of both POEGs produces the same subgraph for all but process 0 (Figure 5.19b), which must be tested for two orderings of incoming messages.

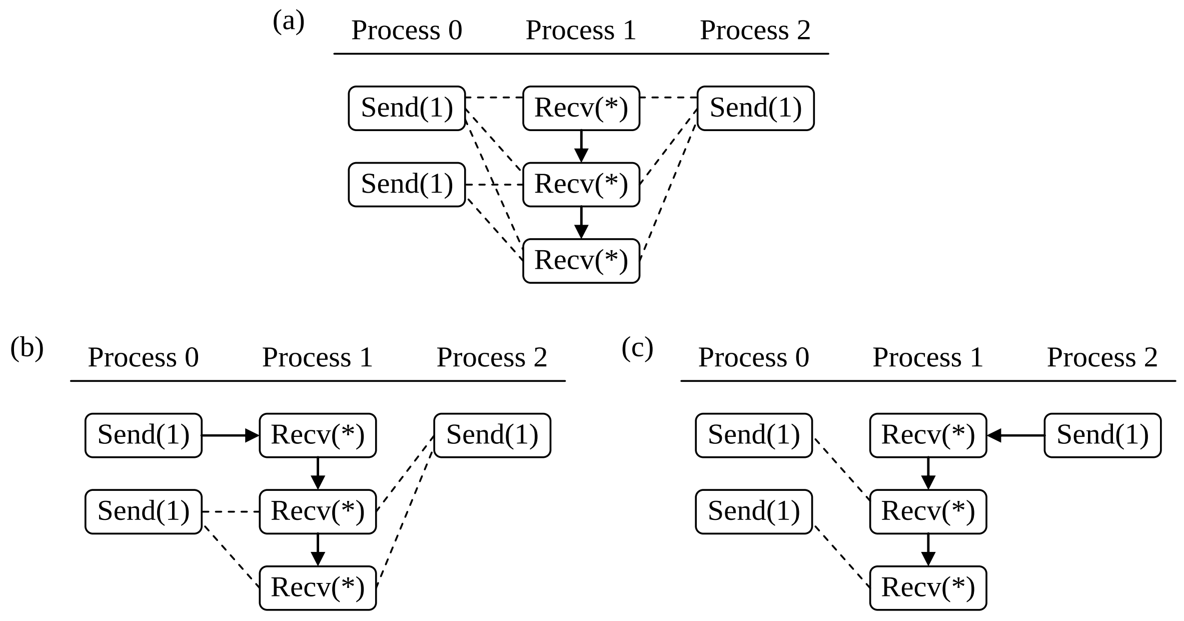


Figure 5.18: *(a) Each wildcard receive matches multiple send calls, (b) however the actual possible matches depends on the match of the first wildcard receive; (c) the POEG does not contain information about FIFO communications enabling determining that if the first wildcard receive matches the send from process 2, the other wildcards are actually matched deterministically.*

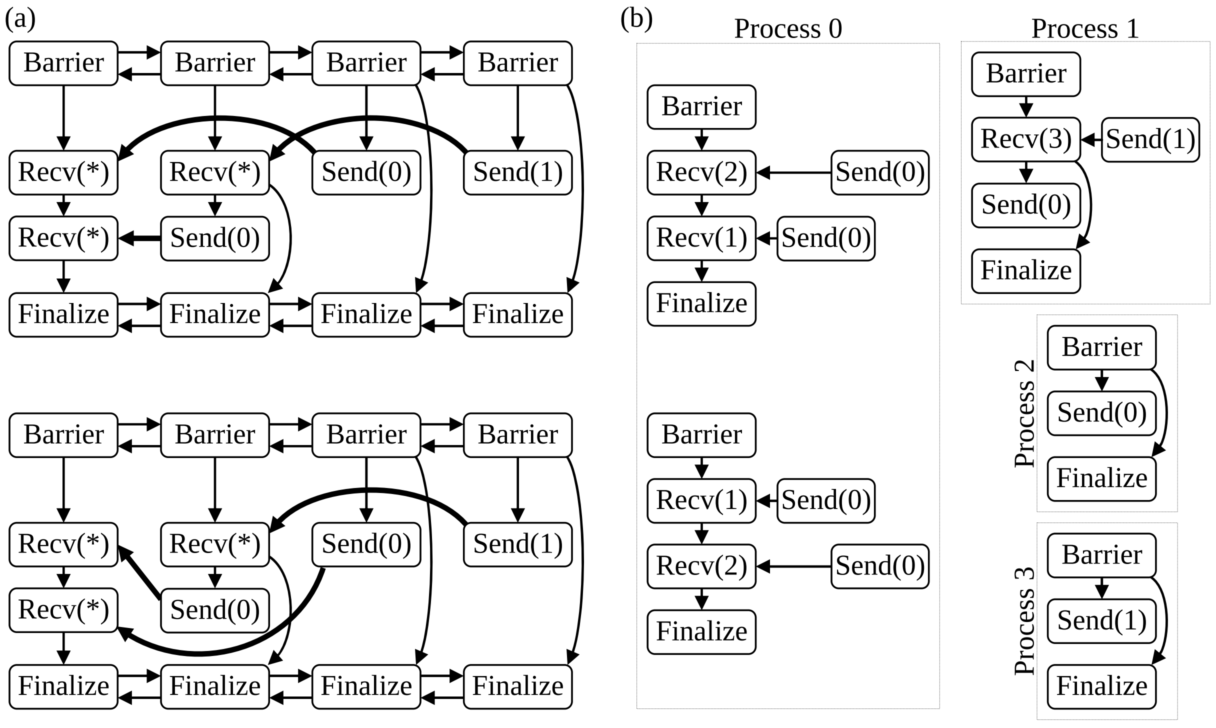


Figure 5.19: *(a) The two possible POEGs representing the possible matches of wildcard receives for the merge sort message-passing graph shown in Figure 5.7; (b) decomposing each POEG along process boundaries provides a set of orderings to be tested for each process.*

5.5.4 Detecting potential performance optimizations

MPI calls providing strong synchronization guarantees usually have lower performance than corresponding calls with looser guarantees. There are therefore potential performance benefits in avoiding using strongly synchronizing calls when they are not strictly necessary. However, it is often difficult to determine whether these guarantees are strictly necessary or not by simply looking at the application code.

The dependencies expressed by the POEG may also help identifying such requirements, thereby enabling potential performance improvements. In this section, we describe how to identify the impact of removing *MPI_Barrier* calls. Barriers are often overused by parallel application writers because they enable clearly separating different phases of the computation. This appears to be the case for instance in Pliris, a dense matrix linear solver of the Trilinos package [46]: removing the barriers greatly reduced the running time (Table 5.1) without impacting the accuracy of the result[3]. Another goal may be the replacement of *MPI_Ssend* by *MPI_Send* calls. A comparison of the performance of synchronous and regular sends for two parallel machines can be found in [91].

Table 5.1: *Running time reduction caused by removing barriers in the Pliris dense matrix linear solver, using one process per compute node. Barriers tend to be proportionally more expensive for smaller matrices and when the number of processes increases.*

Matrix size	8 procs	16 procs	32 procs	64 procs
1000	-67%	-68%	-59%	-61%
10000	-18%	-25%	-30%	-35%
50000	-3%	-6%	-7%	-12%

We illustrate the method using an example taken from the source code of the deadlock detector. Upon creating a new communicator, every process involved sends a *comm_created* notification to the detector (Section 5.5.1). The detector needs to make sure that it has received every notification before any process may send subsequent notifications: this prevents receiving notifications using ranks in the newly created communicator that the detector cannot yet resolve. The required synchronization is implemented by having processes sending *comm_created* notifications using synchronous sends before calling *MPI_Barrier*. Figure 5.20a displays the POEG of an execution on two processes. The detector runs within its own thread on process 0, and receives notifications using wildcard receives. The dashed edges indicate de-

[3]Note that the barriers might still be necessary for preventing message races that did not occur in our tests.

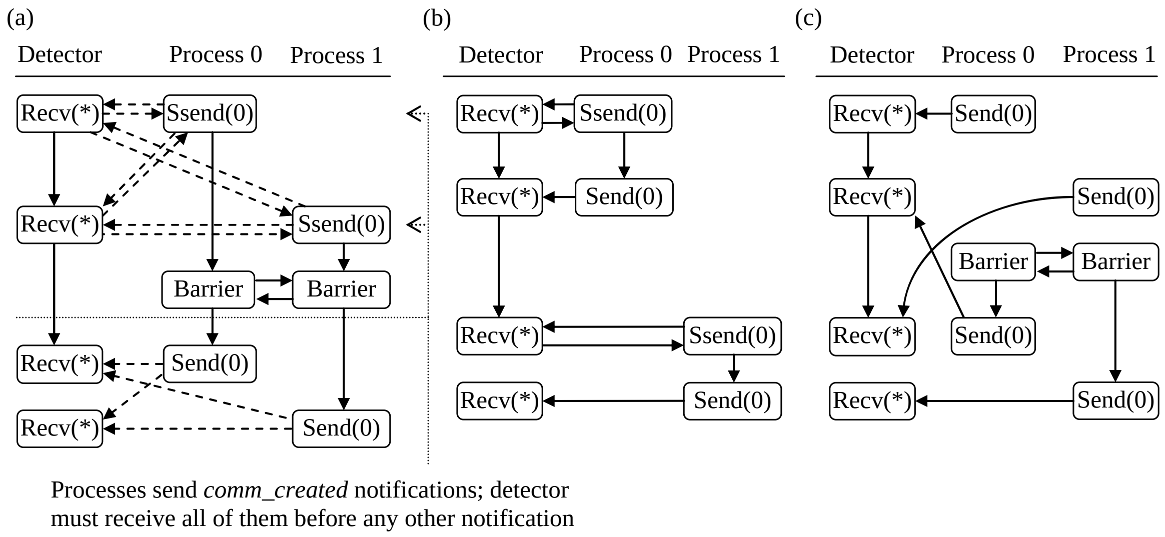


Figure 5.20: *(a) POEG illustrating the creation of a new communicator, when each process sends a* comm_created *notification to the detector using an* MPI_Ssend *call followed by a barrier. Dashed edges indicate dependencies dependent on the send-receive matches; (b) removing the barrier removes the dependency of the send from process 0 on the synchronous send from process 1, and enables executions where regular notifications may be received before all* comm_created *notifications; (c) replacing synchronous sends by regular sends has the same effect, as sends may now be buffered and are no longer predecessors of the barrier.*

pendencies dependent on which send is matched by each wildcard receive. The POEG shows that no matter how wildcard receives are matched, the wildcard receives performed after the barrier are successors of both synchronous sends, i.e. the detector cannot receive any notification before all *comm_created* notifications have been received.

The availability of the POEG makes it easy to determine whether a *MPI_Barrier* call is necessary or not. A barrier call can be safely removed if and only if it does not change the set of send calls matching a wildcard receive. Section 5.20b shows that the barrier is necessary in our example: as the *MPI_Send* from process 0 is no longer a successor of the *MPI_Ssend* from process 1, the detector is no longer guaranteed to receive every *comm_created* notification before any subsequent notification. We can test whether *MPI_Ssend* calls can be safely replaced by *MPI_Send* calls in the same fashion. Figure 5.20c shows that the synchronous sends are indeed necessary to ensure that the wildcard receives called after the barrier cannot match sends called before the barrier.

In practice however, a single function call in the source code can lead to multiple events within the POEG, for instance when the function is called within a loop. The stack trace information of each call must therefore be used to ensure that one can indeed safely remove all instances of barriers or synchronous sends caused by a single source code location. When processes performing different tasks enter a same barrier from different code locations, one must check that all resulting barriers within the POEG can be safely removed.

The approach can be generalized to other calls, or applied to detect specific patterns within the POEG. A sequence of sends of identical size from one process to the others could potentially be replaced by a single *MPI_Bcast* or *MPI_Scatter* call.

5.5.5 Applying message-passing state graphs to MPI applications

We saw in Section 5.5.3 that static POEG representations cannot easily capture the possible executions of an MPI application. Moreover, POEGs are restricted to applications producing a fixed set of MPI events, and cannot represent cases where branches within the application code adapt the execution to the messages that have been received. We therefore present a conceptual generalization of the message-passing state graph construction for MPI applications. In our first example, we only consider applications that perform deterministic computations and use non-buffered blocking point-to-point and collective communications. We therefore want to test that the computation result remains the same no matter which message is actually received by a wildcard receive.

In DPS applications, new computations are triggered by the delivery of a message. In MPI applications that satisfy our restrictions, new computations are triggered when a set of calls from distinct processes are matched. For collective communications, processes block until all participating processes have joined the communication. For blocking point-to-point MPI communications, a send call at the source must match a receive call at the destination before any of the sending and receiving process may resume its computation.

We represent an execution using the same event model as in the previous sections, i.e. every MPI call produces one event. We now construct a message-passing state graph as follows. Starting from the initial application state where all processes have called the *MPI_Init* function, a transition occurs by matching a send event with a receive event, or by matching a set of collective communication events, thereby allowing the suspended processes to resume their execution until the next MPI function call. In respect to the state graph construction, the state of a single process is defined by the value of its local variables and by the pending MPI call. The state of the whole application is defined as the set of states of the individual processes.

We illustrate the state graph construction for the merge sort application presented in Section 5.4.3. We ignore the initial distribution of the array, and assume that each process initially stores a local array containing n/p sorted elements, that must be merged into a single sorted array of size n.

Figure 5.21a recalls the message-passing graph of the expected execution on four processes, when all processes move simultaneously from one iteration to the next. The init and finalize events from process i are denoted I_i and F_i, and are assumed to behave as barrier synchronizations. A send from process i to process j at iteration k is denoted as $s^k_{i,j}$, while $r^k_{*,j}$ denotes the matching wildcard receive event.

If the send $s^0_{2,0}$ of process 2 is delayed, the send event $s^1_{1,0}$ may instead match the $r^0_{*,0}$ receive (Figure 5.21b). As $r^0_{*,0}$ expects an array containing $n/4$ elements while process 1 sends $n/2$ elements, the MPI implementation may either deliver a truncated message or raise a fatal error.

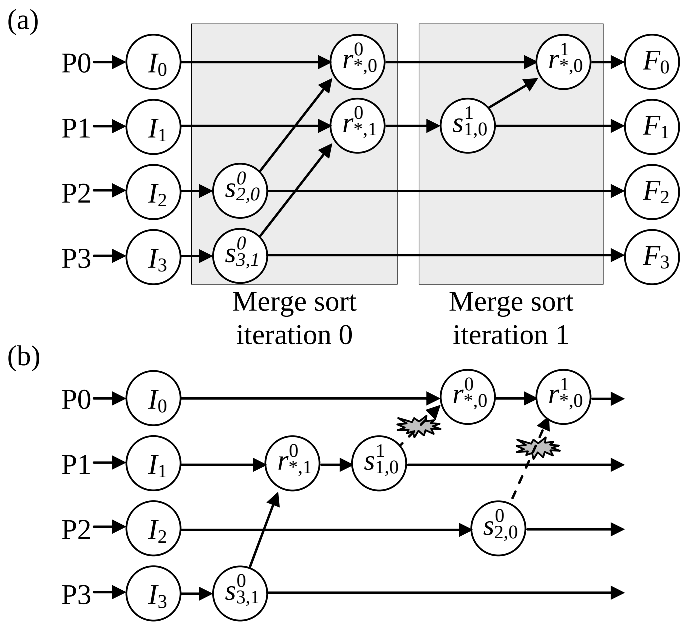


Figure 5.21: *(a) Message-passing graph of a correct execution on four processes and (b) of an incorrect execution, where send from rank 2 is delayed.*

The message-passing state graph of the application is displayed in Figure 5.22a. The match of the send event $s^k_{i,j}$ with the recv event $r^{k'}_{*,j}$ is indicated as $\{s^k_{i,j}, r^{k'}_{*,j}\}$. The incorrect match $\{s^1_{1,0}, r^0_{*,0}\}$ is easily detected as message sizes are different. The programmer can solve the problem by explicitly specifying the source of the expected message in the receive call. Then $s^1_{1,0}$ can no longer match $r^0_{2,0}$ (Figure 5.22b).

Heuristics reducing the number of tested orderings can also be applied. We can for instance adapt the breadth-first and depth-first executions from Section 4.5.5 as follows. We maintain for each process a counter that stores the number of events that have been matched. We then produce a depth-first (resp. breadth-first) execution by always matching events for the process

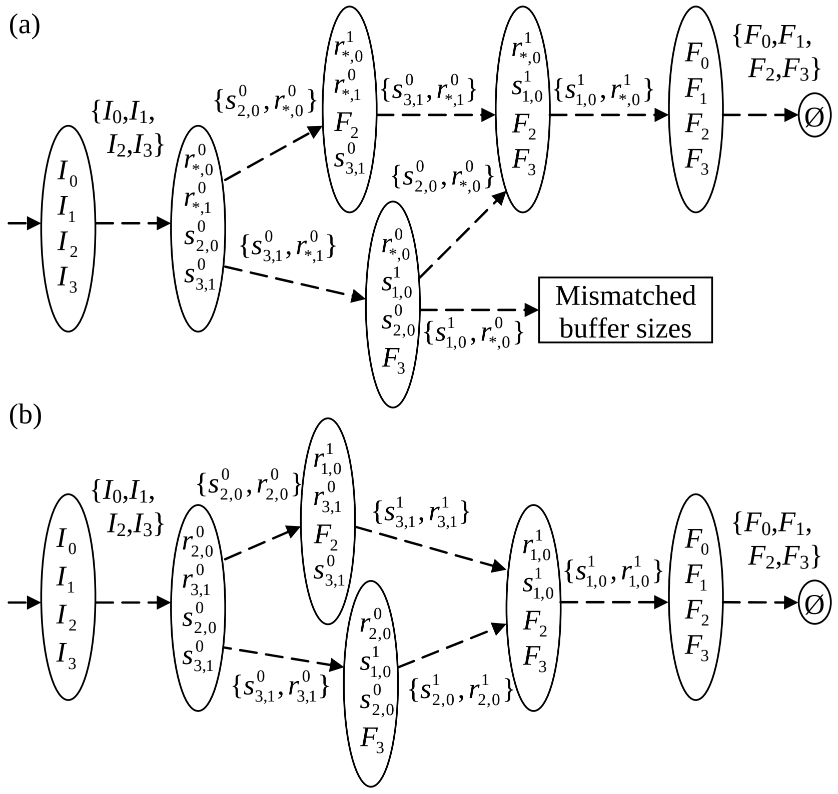

(a)
$\{I_0,I_1,I_2,I_3\}$
$\{s^0_{2,0},r^0_{*,0}\}$
$\{s^0_{3,1},r^0_{*,1}\}$
$\{s^1_{1,0},r^1_{*,0}\}$
$\{s^0_{2,0},r^0_{*,0}\}$
$\{s^1_{1,0},r^0_{*,0}\}$
Mismatched buffer sizes
$\{F_0,F_1,F_2,F_3\}$
Ø
(b)
$\{I_0,I_1,I_2,I_3\}$
$\{s^0_{2,0},r^0_{2,0}\}$
$\{s^0_{3,1},r^0_{3,1}\}$
$\{s^1_{3,1},r^1_{3,1}\}$
$\{s^1_{2,0},r^1_{2,0}\}$
$\{s^1_{1,0},r^1_{1,0}\}$
$\{F_0,F_1,F_2,F_3\}$
Ø

Figure 5.22: *(a) Message-passing state graph of the application with wildcard receives and (b) message-passing state graph without wildcard receives. Nodes indicate the pending event of each process, while edge labels indicate which event match produces the transition.*

with the largest (resp. smallest) counter value. In our example, the error is revealed by a depth-first execution of the application. We initialize every match counter to 0 (Figure 5.23), and after matching the init events we match the events of processes 3 and 1 $\{s^0_{3,1}, r^0_{*,1}\}$. Their match counters are incremented to 2 and become the ones with the largest value. In order to produce a depth-first execution, we therefore attempt to match events from these processes in priority. The finalize event of process 3 (F_3) cannot be matched yet, but we may match the send event $s^1_{1,0}$ of process 1 with the receive event $r^0_{*,0}$ of process 0, thereby producing the incorrect

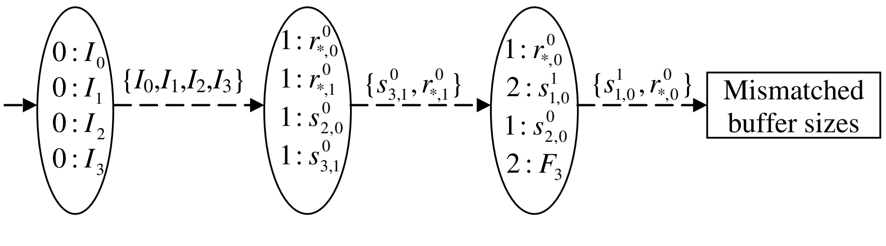


Figure 5.23: *We associate a counter with each process and increment it for every new call completed. Matching the call from the process with the largest counter value produces a depth-first execution.*

execution.

Although one could easily run a single breadth-first or depth-first execution, building the state graph and avoiding restarting the program for each sequence of event matches requires additional checkpointing support. The Berkeley Lab Checkpoint/Restart library has been successfully used with MPI implementations such as LAM and OpenMPI [14, 94]. DéjàVu [92] specifically targets MPI and distributed programs. While these libraries support taking and recovering checkpoints, they do not provide means for performing the fast checkpoint comparisons required for identifying duplicate states. Such capabilities could be provided for C++ MPI applications by the *autoserial* library [99]. The drawback is that the application code must be modified to store its state within a serializable object.

DPS has the advantage of providing a high-level description of dependencies between computations. In MPI, the lack of a flow graph or equivalent information about future communication patterns implies that we cannot readily apply the optimizations described in section 4. However, the potential for optimizations does exist. In Figure 5.22b for instance, all matches are deterministic and both paths in the message-passing state graph are guaranteed to produce the same result. One of the branches could therefore be pruned.

5.5.6 Related work

The testing and verification of MPI programs has made significant progress in the recent years, mostly through the work from two different groups. On one side, Siegel and Avrunin have been working on the development of formal models of MPI applications [106, 107, 108]. Their approach relies on extracting a formal model of an MPI application that can be proved through a model checker such as MPI-SPIN [105]. Their model inspired the adaptation of the POEG and message-passing state graph construction techniques from DPS to MPI applications. They also provide several theoretical results, showing for instance in [106] that if an MPI program using only the *MPI_Send*, *MPI_Recv*, *MPI_Sendrecv* and *MPI_Barrier* functions without wildcards executes once without deadlocking, then all possible executions are deadlock-free. A corollary is that barrier calls are not necessary in such programs [107].

More closely related to our current work is *ISP* (In-Situ Partial Order), which is a combination of an interception library based on PMPI and of a scheduler that controls the execution of the application. ISP actually executes an MPI application for all relevant MPI call orderings and therefore dynamically explores the possible execution space of the application. Equivalent orderings are detected and prevented using a partial-order reduction algorithm. The ideas and algorithms behind ISP are exposed in a series of papers published this year [103, 118, 119, 120, 121].

When an application executes, the ISP scheduler first collects the parameters of all MPI function calls without executing them. This separation between the gathering of function calls information and their execution enables ISP to fully analyze the dependencies between the different calls, potentially enabling their execution in a different order. The execution of MPI calls must however respect the *completes-before* relation between MPI calls; this information is maintained by using intra completes-before (*IntraCB*) edges between MPI calls of a single process, and inter completes-before (*InterCB*) edges between calls from different processes. The resulting graph is very similar to the one we exposed in Section 5.5.3, and also enables detecting *functionally irrelevant barriers* [103] using a procedure similar to the one described in Section 5.5.4.

When deciding upon issuing the MPI calls, the ISP scheduler issues *match-sets* to be executed by the runtime of the different processes. Match-sets specify sets of matching calls (e.g. a send and a receive, or a set of collective calls) similar to the ones we use in Section 5.5.5. ISP however implements no checkpointing, and therefore requires reexecuting the whole application for every ordering.

In order to provide guarantees about which send call is matched by a wildcard receive, *MPI_ANY_SOURCE* parameters are rewritten with the rank of the desired matching process. Note that the idea of rewriting sources was also used by MPI-CHECK [73] to force a wildcard receive to match the send from the process that completed the handshake. The most innovative idea of ISP however is to reorder MPI calls within a single process: the interception library stores non-blocking calls without calling the actual MPI function. The scheduler may then execute the buffered calls in a different order, provided that the completes-before relation is satisfied. The combination of source rewriting and call reordering solves the issue described in Section 5.4.2, where a buffered or non-blocking send hides subsequent sends matching a wildcard receive, as well as the one mentioned in Section 5.5.1 where the deadlock detector may fail detecting deadlocks involving non-blocking wildcard receives.

The graphical user interface displaying the execution of MPI programs (Section 5.4) is complementary to the automatic detection of deadlocks and message races. The ISP team therefore asked for our code and is currently in the process of extending the debugger to display relevant information produced by ISP.

It is worth noting that although we focused on wildcard receives, nondeterminism can occur when using the *MPI_Waitany*, *MPI_Waitsome*, *MPI_Testany* and *MPI_Testsome* calls.

For instance, the *MPI_Waitany* function takes multiple *MPI_Request* objects as parameter and returns as soon as one of the associated non-blocking calls has completed. Assuming that a program runs with two processes, and that both processes send a message to process 0, the call sequence *MPI_Irecv(from 0, req0) – MPI_Irecv(from 1, req1) – MPI_Waitany(req0, req1) – MPI_Waitany(req0, req1)* is semantically equivalent to the sequence *MPI_Recv(*) – MPI_Recv(*)*.

Both ISP and the model-checking approach of Siegel and Avrunin suffer from scalability problems due to the number of MPI call orderings that must be tested. We encountered the same problem in the previous chapter when testing DPS applications. The three approaches therefore rely on the so-called *small-scope hypothesis* [51]: they assume that defects exist for a small number of processes and for small application instances, but remain hidden due to the fact that regular application runs do not cover all possible executions. However, an exhaustive testing approach is able to uncover the defects that would only appear at a larger scale in practice.

5.6 Conclusion

Building on the DPS application debugger presented in the previous chapter, we have presented a debugger for MPI applications that provides the developer with a graphical view of the current status of the application execution. It dynamically draws the message-passing graph of the application, and graph vertices can be highlighted according to specific criteria in order to ease the analysis. Several types of breakpoints enable controlling the execution of the parallel application. All breakpoints operate at the level of message-passing calls rather than code instructions. They enable the developer to focus on the communication patterns of the application, and provide entry points for attaching a sequential debugger to individual processes. The debugger is also able to run the application such that the developer is able to choose how send and receive calls should be matched in the presence of wildcards.

The ability to influence the application by suspending processes and reordering message matches provides the developer with full control over its execution. This control can be used to execute cases that occur only rarely in practice, for example for testing the presence of message races or deadlocks within the parallel application.

We then explored the possible application of the automated testing concepts developed in Chapter 4, namely the use of Partial Order Execution Graphs and of message-passing state graphs to represent possible executions of parallel applications written on top of MPI. We have shown that due to its static construction, a single POEG cannot represent the possible executions of an MPI application using wildcard receives. Such applications may however be represented using a family of POEGs. The dependencies expressed by each POEG may then be used to identify potential performance improvements. These improvements stem from the relaxation of synchronization guarantees, achieved by replacing or removing specific MPI function calls. We then sketched a dynamic testing method for a subset of MPI calls.

The results presented in this section have appeared in [97, 98, 101].

Chapter 6

Conclusion

We presented our results on the performance prediction and the development of advanced testing tools for distributed memory message-passing parallel applications. The proposed solutions have the common characteristic of being applicable on the actual application code, and do not require the development of any model or formalization from the application developer.

A significant number of parameters influence the performance of a parallel application. Given the high usage level of parallel machines, performing a full range of tests for identifying the parameter combination providing the best performance is very time consuming. At the same time, efficient implementations often limit the use of synchronizations and maximize the overlapping of computations and communications. Such strategies are however error-prone, and the lack of tools for testing the correctness of computations in parallel applications encourages the scientists writing these applications to be conservative in their implementations.

We began in Chapter 2 by describing the Dynamic Parallel Schedules (DPS) parallelization framework, within which most of our results have been integrated. DPS has the benefit of providing a high level of abstraction between the application code and the underlying deployment onto threads and processes. Three defining features enabled the current work. Firstly, DPS is multithreaded by nature, and multiple operations may execute simultaneously within different threads. Secondly, threads and operations may be checkpointed and recovered in order to tolerate faults. Thirdly, the use of a flow graph to describe the parallization of the computation cleanly describes the dependencies and communication patterns between individual serial operations.

The multithreaded and object-oriented construction of the framework was instrumental in enabling the integration of a parallel application simulator in Chapter 3. We identified a small set of processing and networking parameters that characterize the hardware platform onto which the application is running. After parameterizing the hardware platform, the running time of parallel applications can be predicted using direct execution without requiring any change to the application source code. We propose a partial direct execution technique that reduces the execution time and memory consumption of the simulation. Using partial direct execution, the simulation is no longer tied to the platform to be simulated. Simulations may thus run on a desktop computer rather than on the target parallel machine. The proposed parameterization of the application and of the hardware properties enable using the simulator to study the sensitivity of a parallel application to various operating conditions such as the data subdivision granularity, the adopted parallelization strategy and the underlying hardware platform properties. The proposed simulator helps developers identifying the factors having the largest impact on their application's performance, and determining the most suitable cluster hardware configuration.

Chapter 4 focuses on detecting implementation and synchronization errors in DPS applications. We developed a debugger for DPS applications that displays an instantaneous graphical view of the global computation state and is able to control the ordering of message delivery in order to explicitly test specific orderings. We then automated the testing process by leveraging the simulator's ability to control the execution of an application. The checkpointing capabilities of DPS are critical for detecting errors since they enable detecting differences in messages and thread states produced by different executions in a fully automatic manner.

We first use a static approach for reducing the number of orderings to be tested to exhaustively cover all possible executions of an application. This first method is based on a partial-order reduction of the search space and on the decomposition of the application execution into independently testable subparts. Using a static method requires that all dependencies between messages and operations can be captured using a single Partial-Order Execution Graph. This constraint limits the application of the method to applications producing a fixed set of messages, i.e. applications producing the same messages for all delivery orderings. We overcome this limitation through a second approach relying on the dynamic construction of a message-passing state graph expressing possible executions. In the latter case, the flow graph of the DPS application enables identifying and preventing equivalent executions. Both methods reduce the testing costs by several orders of magnitude, and can be combined to further improve the results. Nevertheless, testing times may remain prohibitive for longer running applications. We therefore also define algorithms generating subsets of possible orderings that are likely to reveal erroneous executions.

Chapter 5 then focuses on adapting the message race and deadlock detection techniques of

Chapter 4 to applications written on top of libraries implementing the Message Passing Interface (MPI) standard. We first described the extension of our work on visualizing the execution of parallel applications. We then discussed the limits and the benefits of using partial-order execution graphs to describe MPI application executions, and showed how our dynamic message-passing state graph construction approach can be applied. The relevance of our approach is then assessed through a comparison with very recent related work.

On the ease of use of the proposed methods

Simulating a DPS application requires at least two steps. The first is to parameterize the target cluster hardware, e.g. using the provided parameterization tool (Appendix A). The second step is to recompile the application with an additional preprocessor definition. However, the resulting simulations will likely require excessive amounts of memory and of running time to complete. The proposed partial direct execution scheme imposes that the developer manually replaces expensive computations with simulator notifications containing duration estimates. This second phase may require significant work, but has two major benefits. Firstly, forcing the developer to identify the most expensive computations improves his understanding of his application. Secondly, the result is a fully parameterized application, which can be further analyzed by varying the duration of specific computations and communications.

Enabling the automated message race and deadlock detection in a DPS application requires always initializing members of serializable objects and making threads and operations checkpointable. In most cases this process is fairly easy, and it is equivalent to making the application fault-tolerant. Once this is done, the only remaining task for the developer is to determine whether diverging executions reported by the validator indeed reveal the occurrence of message races or whether they are instead fully in accordance with the expected execution of the application.

For MPI applications, our analysis only relies on the list of MPI calls, including their parameters, performed by an application. The use of MPI's built-in profiling interface enables obtaining that information by simply linking an interception library to the MPI program, and is therefore transparent for the application developer.

Future work

The major limitation of the simulator is its lack of support for multicore processors. The integration of a simple model taking into account resource sharing on the memory bus would enable accurate predictions on today's clusters of multicores. The simulation of multiple simultaneously executing applications would provide another interesting research direction. The dynamic thread and process allocation features of DPS, the simulator's initial multi-application simulation support, and its ability to produce detailed runtime statistics about CPU and network resource usage may be leveraged and extended to determine the optimal scheduling of a set of applications on a set of processing nodes. Such research would enable maximizing the utilization of parallel machines, potentially reducing the turnaround time of parallel jobs by compensating the lower efficiency of some applications.

Regarding the detection of errors in DPS applications, the exhaustive exploration of relevant executions is only applicable to small application instances due to the combinatorial explosion of the number of possible message orderings. While we proposed heuristics for testing subsets of possible executions, the static decomposition method also enables differentiating the testing effort applied to different parts of an application execution. However, making such decisions manually becomes unpractical when the number of parts increases. Being able to automatically determine which locations are more likely to contain synchronization errors and require more extensive testing would greatly help developers focus their effort.

The testing of MPI applications would be much improved by the availability of built-in checkpoint/restart capabilities. Their integration within the visualization GUI would for instance enable testing execution variants interactively. Similarly, the ISP team is currently working on integrating their message race and deadlock detection tool and our visualization program. However, the time needed for dynamically exploring relevant executions would be dramatically reduced if checkpoints could be used to avoid restarting the tested program from the beginning for every execution.

Appendix A

Simulator tools

A.1 Cluster parameterization

The simulator requires a small set of cluster-specific parameters. These are namely the latency and the bandwidth of the network, and the amount of CPU consumed by network transfers as a function of the number of simultaneously outgoing and incoming transfers. The DPS distribution includes a tool, *clusterParams* that automatically measures these values and produces cluster configuration files that serve as an input to the simulator.

The latency and bandwidth are measured using simple ping-pong benchmarks. The latency is measured as the time required to send an empty serializable object between two DPS operations. The bandwidth is given by dividing the size of very large messages by the time needed to transfer these messages. CPU consumption is evaluated by measuring the slowdown of a CPU intensive computation while network transfers are being carried out.

Since *clusterParams* is a regular DPS application, all measurements include overheads due to the framework and to the serialization of messages. The measured latency is therefore slightly larger and the bandwidth slightly lower than for raw data transfers but these numbers correspond to the values effectively seen by DPS applications. However, *clusterParams* only transfers simple objects, and its measurements do therefore not include the higher latency and CPU utilization produced by the serialization of complex objects (see Appendix B).

A.2 Automated benchmarking

DPS provides a few helper classes to automate the process of taking and reusing benchmark measurements.

A.2.1 Collecting measurements

Measurements are collected using the *BenchWriter* class. The file to which measurements are appended is specified as a parameter to the *BenchWriter::init* method. Cleanup is performed by calling *BenchWriter::finalize*.

```
// Before the first measurement is taken
dps::sim::BenchWriter::get()->init("measures.csv");
// Before the application terminates
dps::sim::BenchWriter::get()->finalize();
```

The *BenchWriter* class acts as a stop watch, where the timer is started by calling *start*, and calling *write* causes the time elapsed since the last call to start to be written to the file specified when init was called. The write method may take between one and four double parameters in addition to the name that identifies the measured function. The function name and the parameters form a single key that identifies the measurement. The parameters of *write* should therefore be the parameters that influence the duration of the computation enclosed within the calls to *start* and *write*.

```
dps::sim::BenchWriter::get()->start();
// Do things during 700000 microseconds
dps::sim::BenchWriter::get()->write("funcName",12,3.45);
```

The following line is then added at the end of the output file:

```
funcName,12.000,3.450,700000
```

Instrumenting the application code enables collecting measurements by running the application as usual, and automatically causes more measurements to be taken for frequently occurring computations. The instrumented application may be run within the simulator to guarantee that no two operations ever run simultaneously, and that writes to the file never overlap.

A.2.2 Reusing measurements

The process of retrieving the measurements from a data file is quite similar, and is handled through the *Bench* class. The call to init aggregates the data read in the specified file by averaging all the measurements for every key, i.e. for every tuple (*function name*, *param1*, ...).

The timings may then be retrieved using the *Bench::getTiming* method, which takes the same parameters as *BenchWriter::write*.

```
dps::sim::Bench::get()->init("measures.csv");
double t=dps::sim::Bench::get()->getTiming("funcName",12,3.45);
// t equals 700000
```

A.3 Instrumentation example

The LU factorization application described in Chapter 3 is included within the DPS distribution, and the *lu-sim* version includes benchmarking capabilities. When the application is compiled for simulations, command line parameters can be used to control its running mode: *-directExec* produces a simulation using direct execution simulation, and *-simBench* collects operation running times enabling partial direct execution. If no option is specified, the simulation runs in partial direct execution using previously collected running times. We illustrate the instrumentation enabling these multiple running modes using the *InitialSplit* operation.

The *InitialSplit* operation spends most of its time performing the LU decomposition of a

Listing A.1: *Original implementation of the* InitialSplit *operation of the LU factorization application.*

```
void execute(RoutedDataObject *in)
{
  LUApp *app=(LUApp*)getApplication();
  if(in) {
    // Initialize r x r square matrix block with uppermost part of the
    // block column stored in thread (localMat)
    Matrix sm;
    sm.set(getThread()->localMat,0,0,r,getThread()->localMat.rows);

    if(sm.cols%SBSIZE!=0)          // Perform LU factorization of sm
      ludcmp(sm,getThread()->vv);  // vv contains pivoting information
    else
      blockludcmp(sm,getThread()->vv,SBSIZE);
  }

  for(int i=1;i<getThread()->blocksX;i++) {
    MatrixDataObject *lur=new MatrixDataObject();
    lur->matrix.set(getThread()->localMat,0,0,r,r);
    lur->route=i;
    lur->level=0;
    lur->blocksX=getThread()->blocksX;
    postDataObject(lur,i);
  }
}
```

block stored in the local state (*ludcmp* or *blockludcmp*). We therefore remove this operation, while the loop that posts the output data objects is left as is. The running times of both *ludcmp* and *blockludcmp* depend on the parameters *sm.cols* and *sm.rows*. Since the value of *sm.cols* also determines the choice of the decomposition function to be used, we do not need to distinguish the two functions in the measurement file. The code below assumes that *SBSIZE* is a constant parameter, however it could be easily added as a third parameter.

Listing A.2: *Instrumented version of* InitialSplit*. The same code is used for benchmarking and for running direct and partial direct execution simulations.*

```
void execute(RoutedDataObject *in)
{
  if(in) {
    Matrix sm;
    sm.set(getThread()->localMat,0,0,r,getThread()->localMat.rows);

#ifdef DPS_SIM // Start simulation specific code
    if(bench)
      dps::sim::BenchWriter::get()->start();  // Start timer
    if(bench || directExec)    // Benchmarking and direct execution
    {                          // require executing the real code
#endif
    if(sm.cols%SBSIZE!=0)
      ludcmp(sm,getThread()->vv);
    else
      blockludcmp(sm,getThread()->vv,SBSIZE);
#ifdef DPS_SIM // Start simulation specific code
    } // Close if opened in other #ifdef block
    else           // If we don't benchmark or use direct execution
    {              // we use previous benchmark measurements
      addComputationTime((Int64)dps::sim::Bench::get()->
                          getTiming("ludcmp",sm.rows,sm.cols));
    }
    if(bench) // If benchmarking, record computation duration
      dps::sim::BenchWriter::get()->write("ludcmp",sm.rows,sm.cols);
#endif
  }

  for(int i=1;i<getThread()->blocksX;i++)
  {
    MatrixDataObject *lur=new MatrixDataObject();
    lur->matrix.set(getThread()->localMat,0,0,r,r);
    lur->route=i;
    lur->level=0;
    lur->blocksX=getThread()->blocksX;
    postDataObject(lur,i);
  }
}
```

A.3.1 Avoiding memory allocations

The *lu-sim* application also supports the *-noAlloc* parameter, which instructs the application not to allocate matrices during the execution of the program. The example below shows how the multiplication operation was modified to support this running mode. It uses the *simResize* function of the *Matrix* class, which fakes the allocation of the matrix without changing the apparent size of the object to the simulated network layer.

Listing A.3: *Instrumented version of* MulMultiply *matrix multiplication operation.*

```
void execute(DualMatrixDataObject *in)
{
  Matrix rm;
#ifdef DPS_SIM // Start simulation specific code
  if(bench)
    dps::sim::BenchWriter::get()->start();
  if(bench || directExec)
#endif
  matmul(in->matrix1,in->matrix2,rm);
#ifdef DPS_SIM // Start simulation specific code
  else
  {
    // Allocate memory as matmul would do
    if(noAlloc) // Pretend to allocate the memory
      rm.simResize(in->matrix2.cols,in->matrix1.rows);
    else        // Perform actual allocation
      rm.resize(in->matrix2.cols,in->matrix1.rows);
    // Retrieve execution time from previous measurements
    addComputationTime((Int64)dps::sim::Bench::get()->
                       getTiming("matmul",
                                 in->matrix1.rows,
                                 in->matrix1.cols,
                                 in->matrix2.cols));
  }
  if(bench)
    dps::sim::BenchWriter::get()->write("matmul",
                                        in->matrix1.rows,
                                        in->matrix1.cols,
                                        in->matrix2.cols);
#endif
  // Fill in and send the computation result
  MatrixDataObject *mt=new MatrixDataObject();
  mt->matrix=rm;              mt->route=in->xOff;
  mt->xOff=in->xOff;          mt->yOff=in->yOff;
  mt->level=in->level;        mt->blocksX=in->blocksX;

  postDataObject(mt);
}
```

Appendix B

Autoserial performance

B.1 Experiment

We briefly present a few performance results showing the performance of the serialization mechanism for a set of serializable objects. For container objects such as buffers or vectors, measures were taken for various sizes. All measurements were taken on the computer used to produce the results of Chapter 4. The serializable objects used are:

Contiguous memory objects

IntDouble contains an *int* and a *double*.

Misc contains three *int* and three *double*.

IntBuffer An object containing an integer and a buffer of integers.

Buffer<Misc> A *Buffer* object containing *Misc* objects.

Non-contiguous memory objects

StdVector A C++ *std::vector* containing integers.

LinkedList A linked list, each node contains an integer and a pointer to the next element of the list.

The *serialization* benchmark consists in serializing the object within an *OpaqueObject*. Opaque objects provide a simple way for serializing an object within a memory buffer. The serialization time includes the creation of the opaque object and the time needed to serialize the object of interest.

```
autoserial::OpaqueObject opaque;
opaque.set(object);
```

The *deserialization* benchmark consists in constructing an object stored within an opaque object. The measurement includes the time needed to deserialize the object and delete it.

```
autoserial::ISerializable *obj = opaque.get();
delete obj;
```

The *comparison* benchmark consists in comparing the original object with a clone produced by storing and recovering it from an opaque object. In order to prevent the compiler to optimize away the comparison, we increment a counter if the objects are different. Since we compare identical objects, the counter is never incremented.

```
if (!obj->equals(clone))
  count++;
```

B.2 Results

The results below display the serialization or comparison rate according to two metrics. The first rate, in megabytes per second, gives an idea of the efficiency of the serialization. The second rate, in objects per second, gives an idea of the number of operations that can be performed per time unit. This is particularly useful for estimating the cost of taking and comparing checkpoints in Chapter 4. The results below show that for our applications, which use serializable objects similar to *IntBuffer* or to *Buffer<Misc>*, it is possible to take and compare several thousands thread state and message checkpoints every second.

Table B.1: *Serialization, deserialization and object comparison rate in objects per second. For reference, the effective serialization rate in MB/s is 8 for* IntDouble *objects, and 13MB/s for* Misc *objects.*

	IntDouble (34 bytes)	*Misc* (53 bytes)
Serialization	$2{\cdot}10^5$	$2{\cdot}10^5$
Deserialization	$1{\cdot}10^6$	$1{\cdot}10^6$
Comparison	$5{\cdot}10^5$	$3{\cdot}10^5$

Table B.2: *Serialization, deserialization and object comparison rate in objects per second. For reference, the effective serialization rate in MB/s is 15 for* IntBuffer *objects containing 10 integers (1.9GB/s for buffers of 10000 integers), and 98MB/s for* Buffer<Misc> *objects containing 10* Misc *objects (1.1GB/s for buffers of 10000 elements).*

	IntBuffer		Buffer<Misc>	
	10 elements	10k elements	10 elements	10k elements
	66 bytes	40.026kB	509 bytes	480.029kB
Serialization	$2{\cdot}10^5$	$5{\cdot}10^4$	$2{\cdot}10^5$	$3{\cdot}10^3$
Deserialization	$1{\cdot}10^6$	$9{\cdot}10^4$	$1{\cdot}10^6$	$3{\cdot}10^3$
Comparison	$5{\cdot}10^5$	$7{\cdot}10^4$	$5{\cdot}10^5$	$4{\cdot}10^3$

Using more complex serializable objects has a large impact on the serialization and comparison time. This is already apparent for the C++ vector (*StdVector*), and it becomes particularly prohibitive when serializing a linked list. Unlike all other container objects for which the efficiency of the occupation of the memory bus increases as the number of elements contained increases, the serialization and comparison performance of the *LinkedList* object decreases as the list grows in size.

Table B.3: *Serialization, deserialization and object comparison rate in objects per second. For reference, the effective serialization rate in MB/s is 15 for* StdVector *objects containing 10 integers (328MB/s for vectors of 10000 integers), and 27.5MB/s for* LinkedList *objects containing 10 nodes (1.2MB/s for linked lists of 10000 nodes).*

	StdVector		LinkedList	
	10 elements	10k elements	10 elements	10k elements
	66 bytes	40.026kB	202 bytes	190.012kB
Serialization	$2{\cdot}10^5$	$8{\cdot}10^3$	$1{\cdot}10^5$	6
Deserialization	$4{\cdot}10^5$	$5{\cdot}10^3$	$2{\cdot}10^5$	$2{\cdot}10^2$
Comparison	$2{\cdot}10^5$	$2{\cdot}10^2$	$7{\cdot}10^4$	3

B.2.1 *AS_MPI_Send* vs. *MPI_Send*

Figure B.1 displays the effective bandwidth of the Pleiades cluster at EPFL when sending buffers using *MPI_Isend–MPI_Irecv* and *MPI_Send–MPI_Recv*. The graph also displays the effective bandwidth achieved by using *AS_MPI_Send–AS_MPI_Recv* with different types of serializable objects, namely *IntBuffer*, *Buffer<Misc>* and *StdVector*. The bandwidth is computed using a ping-pong benchmark sending the same buffer or object back and forth between two processes. Sending buffers of integers or of *Misc* objects reaches nearly the same performance as raw MPI sends when the buffer size reaches 5kB. The small performance drop around 250kB corresponds to the point where the behavior of blocking MPI calls starts diverging from non-blocking ones. On the other hand, the transfer of the C++ vector suffers from the fact that each deserialized element must be pushed back into a new vector at the reception.

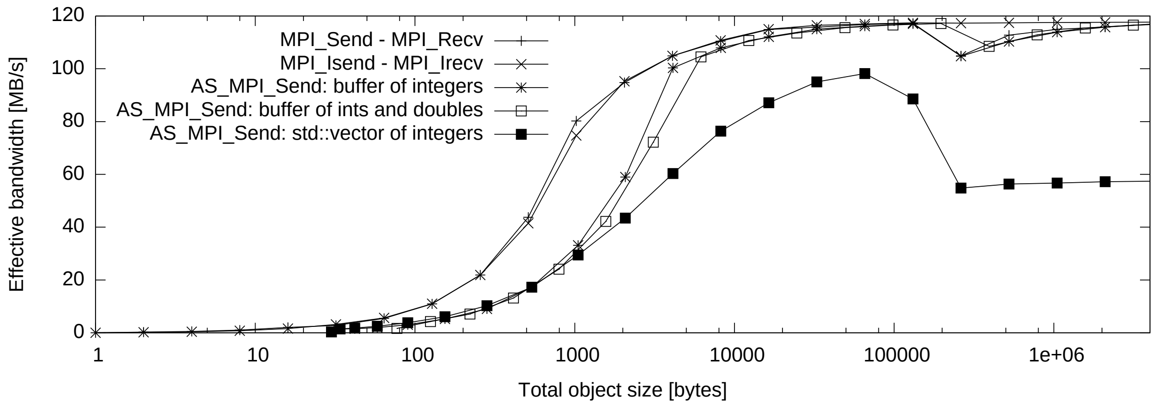


Figure B.1: *Comparison of the effective bandwidth achieved by non-blocking and blocking MPI point-to-point communications, and by the wrappers provided by the* autoserial *library.*

Appendix C

DPS over MPI

C.1 Introduction

We recently implemented an additional network layer within DPS, based on MPI. Since DPS only performs point-to-point communications, it uses only *MPI_Send* and *MPI_Recv* functions. However, the use of MPI within the network layer transforms DPS applications into regular MPI applications, such that the *mpiexec* launcher, the MPI daemon and authentication issues are all handled by MPI. Within the application, MPI also takes care of starting up and shutting down processes, establishing communications between them, and is able to use the faster interconnects that may be available on the target cluster.

This section briefly describes the major implementation issues, and explores potential improvements and issues brought by making DPS and MPI coexist within a single parallel application. More details about the implementation and additional performance results can be found in [31].

C.2 Implementation overview

Most of the implementation is hidden within the network layer, and the use of the MPI or of the TCP network layer is therefore mostly transparent both to the DPS runtime and to the application developer. However, the fact that all processes start their execution simultaneously requires some changes in the initialization procedure of the DPS controller. Upon startup, all controllers duplicate the *MPI_COMM_WORLD* communicator so that message transfers handled by DPS do not interfere with additional MPI communications performed by the user. The major

Table C.1: *User-visible changes in application behavior.*

	DPS over TCP	DPS over MPI
Process identification	*hostname:port*	Rank in *MPI_COMM_WORLD*
Heterogeneous cluster support	Yes	Not implemented, but supported by MPI-2†
Dynamic process creation, destruction and migration	Yes, processes are started as needed	Not implemented, but supported by MPI-2‡
Fault-tolerance	Yes	No, maybe using FTMPI [29]

†Cross-platform serialization is handled by the serialization mechanism and is independent of the network layer.
‡Thread checkpointing and migration is handled independently of the network layer.

differences in functionality are listed in Table C.1.

The only user-visible change, apart from the use of *mpiexec* to start applications, lies in the addressing used to map thread collections onto processes (Section 2.8.1). While the TCP network layer identifies processes using *hostname:port* identifiers, MPI processes are identified using their rank in *MPI_COMM_WORLD*. A *PatternMapper* helper class can be used to automatically generate lists of TCP identifiers from a file containing a list of machines, while the *MPIMapper* helper class automatically generates lists of MPI ranks based on the number of processes in *MPI_COMM_WORLD*.

Due to the asynchronous delivery of messages within DPS applications, each process must be able to send and receive messages simultaneously in order to avoid deadlocks. The MPI implementation must therefore support the *MPI_THREAD_MULTIPLE* level of threading, enabling multiple threads to call MPI functions simultaneously. When TCP is used, the network layer of each process creates one sending and one receiving thread per remote process. Connections are open as needed when messages are sent between processes during the execution of the flow graph. Tests showed that using multiple sender and receiver threads also improved performance when using MPI.

The MPI network layer implementation therefore replaces each *send* call on a TCP socket by an *MPI_Send* call. Since all communications are asynchronous within DPS, there is no need for the stricter synchronization guarantees of *MPI_Ssend*. The rationale for not using non-blocking sends is to avoid the bookkeeping required for correctly completing the non-blocking calls and for deallocating all buffers. While the present implementation may still be improved, it is interesting to note that despite the perfect match between the number of TCP and MPI send calls, the performance of some applications may vary. Figure C.1 for instance illustrates the slowdown incurred when using the MPICH2 MPI implementation on Windows with an embarrassingly parallel DPS application generating a 20,000 $\times$ 20,000 image of the Mandelbrot set. The master sends tasks requesting the computation of a certain number of image lines to each thread, and we compare the overall application running time when requesting 4 and 400 lines per task, with and without activating the flow control mechanism (Section 2.6.2) which limits the number of task requests sent at once by the master.

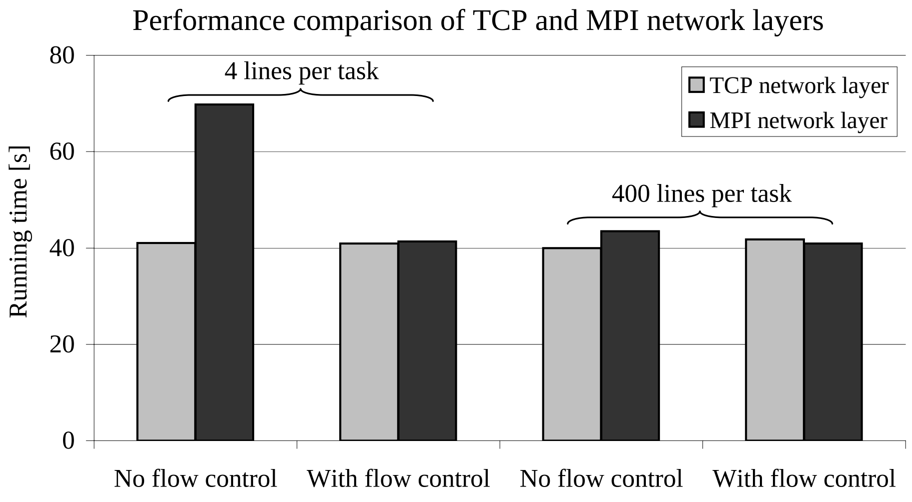


Figure C.1: *Comparison of the running time of a task-farm DPS application when using the TCP and the MPI network layers. Measurements were collected on 4 Windows nodes running MPICH2.*

The running time difference illustrated in Figure C.1 disappears when performing the same comparison using MPICH2 on Linux, i.e. on that platform the TCP and MPI implementations exhibit the same performance. These results therefore show that performance is dependent on the underlying implementation [8, 72, 76, 123].

The performance prediction model that we presented in Chapter 3 may therefore need to be adapted to specific MPI implementations in order to correctly model their impact on the performance of applications. For instance, MPISIM [87] takes into account the exchange of a *request-to-send* message and a *ready-to-receive* reply between the sender and the receiver before actually transferring a message.

Nevertheless, the overall performance of the MPI implementation is quite good, as illustrated by a performance comparison of HPL [25] and of an optimized DPS implementation [35] of the LU factorization (Figure C.2). The performance gap is wider for matrices smaller than 5000 $\times$ 5000, where HPL is up to 30% faster.

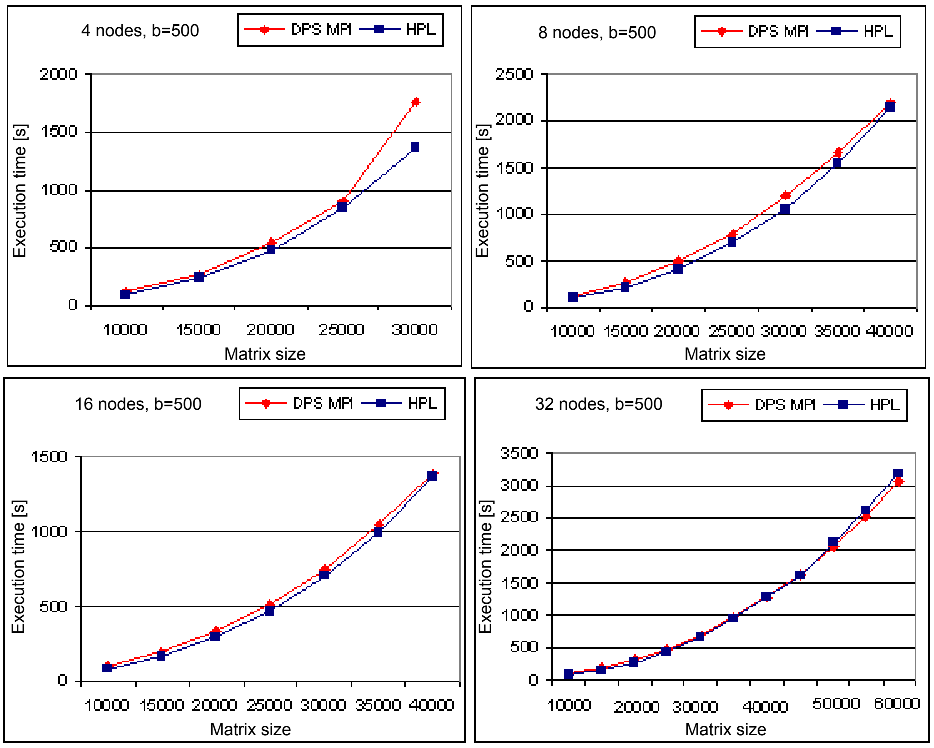


Figure C.2: *LU factorization running time vs. matrix size on 4, 8, 16 and 32 nodes (source: [31]). These compare the running times of HPL vs. those for a block-cyclic implementation of the LU factorization using DPS ([35], Section 4.4.5). The performance gap is wider for smaller matrices, where HPL is up to 30% faster.*

C.3 Enabling new capabilities

The presence of MPI as the underlying communication layer enables developers to call MPI functions from within DPS operations. The most significant is that it enables the use of existing parallel libraries such as ScaLAPACK [19, 95] and FFTW [33].

However, using MPI calls within DPS operations violates many properties enabling the deadlock-free execution of DPS flow graphs[1]. For instance, the implementation of the DPS runtime guarantees that if an operation A depends on the execution of another operation B, B will be able to execute even if it has to run on the same thread that executes A. This happens when split, stream and merge operations are suspended while expecting an input message or a flow control notification (Section 2.5.2). In contrary, calling a blocking MPI or parallel library function stops the operation execution without releasing the thread for other operations.

Other difficulties arise from the fact that DPS and MPI do not address message destina-

[1]The direct use of MPI functions in a DPS application also prevents simulating (Chapter 3) and detecting message races (Chapter 4) in that application.

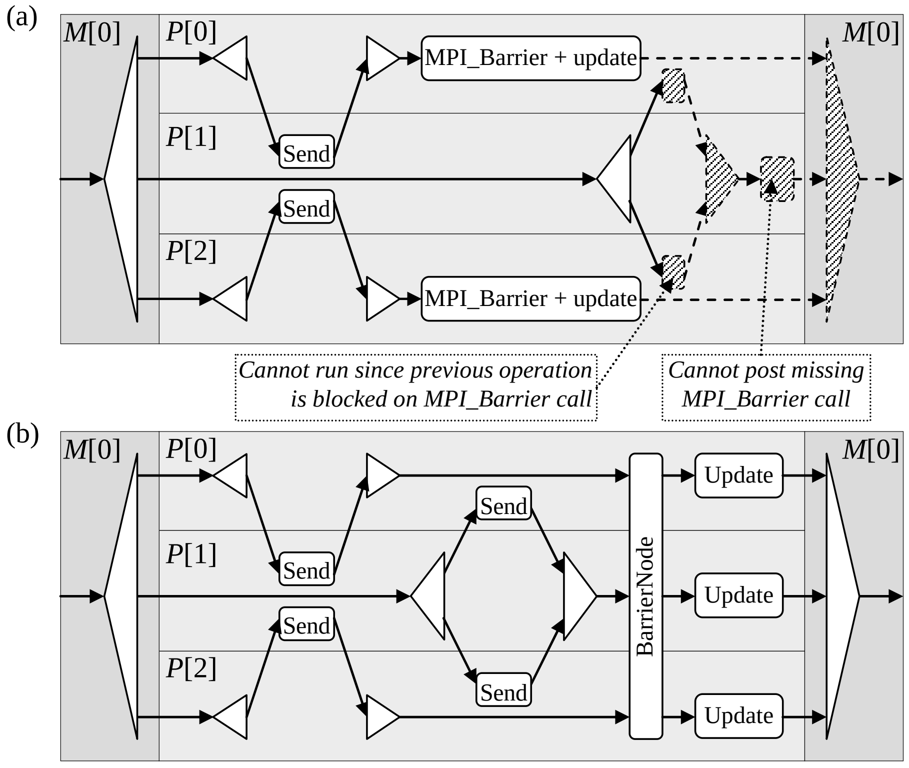


Figure C.3: *(a) Calling* MPI_Barrier *suspends the execution of the underlying DPS thread, preventing the execution of the striped operations; (b) this issue is avoided by using a proper* BarrierNode *wrapper.*

tions in the same fashion: DPS addresses destinations according to a *thread* index, while MPI uses *process* ranks for that purpose. While this is mostly an issue for point-to-point communications, the fact that collective communications require the participation of all processes of a communicator impose that DPS operations participating in the collective must run within different processes. The correct execution of the application therefore becomes dependent to the mapping of threads onto processes.

Nevertheless, it is possible to abstract MPI operations within constructs that satisfy the original data-driven execution model of DPS. Thread collections and communicators have slightly overlapping functionality in the sense that both represent a set of processing elements that may execute simultaneously. However, a single process may belong to multiple communicators, while a single thread may only belong to a single thread collection. We may therefore associate a communicator to a thread collection provided that all threads are mapped onto distinct processes[2].The attached communicator can then be retrieved by DPS operations in order to perform collective communications, and these operations can be abstracted within specialized

[2]This can be easily enforced by using a special type of thread collection, e.g. created using a method such as *createMPIThreadCollection*.

flow graph nodes.

The next section describes the implementation of operations wrapping *MPI_Barrier* and *MPI_Allreduce*. Such operations may then be used within flow graphs like any other. Figure C.3 illustrates the use of a barrier to solve the synchronization problem encountered in the neighborhood-dependent computation described in Section 4.2. In Figure C.3a, the use of an *MPI_Barrier* directly within an operation leads to a deadlock if the split operation execution on thread $P[1]$ is delayed. This problem is solved by using a *BarrierNode* (Figure C.3b), which ensures that the barrier does not prevent the execution of further operations on the threads $P[0]$ and $P[2]$.

C.3.1 Implementation examples

Listing C.1 illustrates how to use MPI functions to transfer a buffer between two DPS operations. The operation running on the thread with index 1 sends a message to the operation running on the thread of index 0.

Listing C.2 illustrates a possible implementation of an operation wrapping the *MPI_Barrier*

Listing C.1: *Use of* MPI_Send *and* MPI_Recv *within a DPS operation. The* getRankOfThread *helper function is used to translate DPS thread indices into MPI ranks.*

```
class Transfer : public dps::LeafOperation<InMessage, OutMessage>
{
  IDENTIFY(Transfer);
public:
  void execute(InMessage *m)
  {
    // Thread 1 sends an MPI message to thread 0
    int srcThread = 1, destThread = 0;
    if (getThreadIndex() == srcThread)
    {
      int dest = getRankOfThread(destThread, MPI_COMM_WORLD);
      MPI_Send(buf, count, datatype, dest, destThread, MPI_COMM_WORLD);
    }
    else if (getThreadIndex() == destThread)
    {
      MPI_Status status;
      int source = getRankOfThread(srcThread, MPI_COMM_WORLD);
      MPI_Recv(buf, count, datatype, source, destThread,
               MPI_COMM_WORLD, &status);
    }
    postDataObject(new OutMessage());
  }
};
```

Listing C.2: *Possible implementation of a* BarrierOperation. *While it makes sense to represent it graphically as a box similar to a stream operation, it is actually implemented as a leaf operation. The* BarrierOperation *receives an input message, calls* MPI_Barrier *on the communicator attached to the thread collection, and posts the input message as its output. To avoid deadlocks, the operation must run on a distinct operating system thread, or* executor, *attached to the same DPS thread*

```
1   // Barrier operation
2   template<typename tokensT> class BarrierOperation
3     : public LeafOperation<tokensT,tokensT>
4   {
5     // We can't use IDENTIFY for templated operations
6     TEMPLATEDEF1(BarrierOperation,tokensT);
7   public:
8     void execute(tokensT* in)
9     {
10      // Perform barrier on communicator attached to collection
11      MPI_Barrier(this->getComm());
12      in->addRef();
13      postDataObject(in);
14    }
15  };

17  // Barrier operations are added during flow graph construction
18  graphBuilder = ... >>
19    // The dps::NoRoute routing function guarantees that the index of
20    // the destination thread is the same as the source
21    // The executorIndex identifies a distinct operating system thread
22    // for running the operation
23    dps::FlowgraphNode<dps::NoRoute,BarrierOperation<MessageType> >
24        (threadCollection,executorIndex) >>
25    ...;
```

function. The operation is then added to the flow graph (line 24). In order to avoid deadlocks such as the one illustrated in Figure C.3, such operations must be run within a distinct operating system thread[3]; the suspension of the operation therefore does not prevent other DPS operations from running. Other synchronizing collective communications or calls to parallel libraries can be encapsulated in a similar fashion. Since these functions require extra parameters, these must be enclosed within a the input message of the operation. Listing C.3 illustrates the implementation of an operation wrapping the *MPI_Allreduce* function.

The semantics of fan-out and fan-in collective communications are closer to those of the split and merge operations of DPS. In both cases, one process or thread distributes data to, or collects data from multiple others. The problem here is that a split typically executes on a different thread collection from the one to which its outputs are delivered. This behavior may

[3]See Section 5.2.3 in [35] or the Basic Tutorials on the DPS website [37].

Listing C.3: *Possible implementation of an* AllreduceOperation. *The use of template arguments lets the developer provide his own message type provided that it implements the* getAllreduceInfo *method, which is used to recover an* AllreduceInfo *object containing the parameters by the* MPI_Allreduce *function.*

```
// Encapsulate parameters of MPI_Allreduce
class AllreduceInfo
{
public:
  void *inbuffer, *outbuffer;
  int count, root;
  MPI_Datatype datatype; MPI_Op op;
};

// AllReduce operation
template<typename tokensT> class AllreduceOperation
  : public LeafOperation<tokensT,tokensT>
{
  TEMPLATEIDENTIFY1(AllreduceOperation,tokensT);
public:
  void execute(tokensT* in) {
    AllreduceInfo &info = in->getAllreduceInfo();
    MPI_Allreduce(info.inbuffer, info.outbuffer, info.count,
                  info.datatype, info.op, this->getComm());
    in->addRef(); postDataObject(in);
  }
};
```

be emulated using a pair of DPS operations: a split broadcasts small notifications to all threads of the destination collection, which trigger a leaf operation that calls *MPI_Bcast*.

Such wrapper operations can be made more easy to use by extending already existing objects within the DPS framework. On may for instance write a *BarrierNode* class that extends *FlowgraphNode* to take care of the routing function and of the use of dedicated operating system threads. For fan-in and fan-out collective communications, partial flow graph constructions can be wrapped within a *FlowgraphSection* object[4]. Such tighter integration within the DPS framework provides stricter semantics for wrapped MPI and parallel library functions, thereby enabling their integration within the simulation and message race detection techniques described in this thesis.

[4]See Section 3.5.9 in [35] or the Advanced Tutorials on the DPS website [37].

Bibliography

[1] Vikram S. Adve and Mary K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94–136, February 2004.

[2] Vishwani D. Agrawal and Srimat T. Chakradhar. Performance estimation in a massively parallel system. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 306–313, Washington, DC, USA, 1990. IEEE Computer Society.

[3] Ali Al-Shabibi, Sebastian Gerlach, Roger D. Hersch, and Basile Schaeli. A debugger for flow graph based parallel applications. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 14–20, New York, NY, USA, 2007. ACM.

[4] Cosimo Anglano. Predicting parallel applications performance on non-dedicated cluster platforms. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 172–179, New York, NY, USA, 1998. ACM.

[5] Fabrice Armougom, Sébastien Moretti, Olivier Poirot, Stéphane Audic, Pierre Dumas, Basile Schaeli, Vladimir Keduas, and Cédric Notredame. Expresso: automatic incorporation of structural information in multiple sequence alignments using 3d-coffee. *Nucleic Acids Research*, 34(Web-Server-Issue):604–608, 2006.

[6] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G.L. Lee, B.P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[7] Rajive Bagrodia, Ewa Deeljman, Steven Docy, and Thomas Phan. Performance prediction of large parallel applications using parallel simulations. In *PPoPP '99: Proceed-*

ings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 151–162, New York, NY, USA, 1999. ACM.

[8] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. Toward efficient support for multithreaded mpi communication. In *EuroPVM/MPI '08: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 120–129. Springer, 2008.

[9] Susanne M. Balle, Bevin R. Brett, Chih-Ping Chen, and David LaFrance-Linden. Extending a traditional debugger to debug massively parallel applications. *J. Parallel Distrib. Comput.*, 64(5):617–628, 2004.

[10] Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, and Enrique S. Quintana-Ortí. Evaluation and Tuning of Level 3 CUBLAS for Graphics Processors. In *Proceedings of the 22nd IEEE Parallel and Distributed Processing Symposium (IPDPS'08), 9th Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'08)*, 2008.

[11] F. Blagojevic, A. Stamatakis, C.D. Antonopoulos, and D.S. Nikolopoulos. RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[12] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 2.5*. May 2005.

[13] Paolo Bonzini and Laura Pozzi. Code transformation strategies for extensible embedded processors. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 242–252, New York, NY, USA, 2006. ACM.

[14] A. Bouteiller, G. Bosilca, and J. Dongarra. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 297–306. Springer, 2007.

[15] R. Brightwell, D. Doerfler, and K. D. Underwood. A comparison of 4x infiniband and quadrics elan-4 technologies. In *CLUSTER '04: Proceedings of the 2004 IEEE Interna-*

tional Conference on Cluster Computing, pages 193–204, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative Evaluation of the Robustness of DAG Scheduling Heuristics. Technical Report TR-0120, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, December 2007.

[17] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *CAV '07: Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 240–253. Springer, 2007.

[18] G. Chillariga and B. Ramkumar. Performance prediction for portable parallel execution on MIMD architectures. *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 630–634, Apr 1995.

[19] Jaeyoung Choi, Jack J. Dongarra, Susan L. Ostrouchov, Antoine P. Petitet, David W. Walker, and Clint R. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Sci. Program.*, 5(3):173–184, 1996.

[20] Jong-Deok Choi and Sang Lyul Min. Race frontier: reproducing data races in parallel-program debugging. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 145–154, New York, NY, USA, 1991. ACM.

[21] W. Cirne and F. Berman. A model for moldable supercomputer jobs. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 8 pp.+, 2001.

[22] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair, and S. Madala. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[23] James Cownie and William Gropp. A standard interface for debugger access to message queue information in mpi. In *EuroPVM/MPI '99: Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 51–58, London, UK, 1999. Springer-Verlag.

[24] M. Crovella, R. Bianchini, T. LeBlanc, E. Markatos, and R. Wisniewski. Using communication-to-computation ratio in parallel program design and performance prediction. *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*, pages 238–245, Dec 1992.

[25] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[26] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Technical Report 54941, Lawrence Berkley National Laboratory, 2002.

[27] E. Duesterwald, R. Gupta, and M. L. Soffal. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 1992.

[28] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976.

[29] Graham E. Fagg and Jack J. Dongarra. Building and using a fault-tolerant mpi implementation. *Int. J. High Perform. Comput. Appl.*, 18(3):353–361, 2004.

[30] Eitan Farchi, Shmuel Ur, Yarden Nir-Buhcbinder, and Orit Edelstein.

[31] Mamy Fetiarison. DPS over MPI. Master's thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), 2008.

[32] C. Flanagan, S.N. Freund, and S. Qadeer. Exploiting purity for atomicity. *Software Engineering, IEEE Transactions on*, 31(4):275–291, April 2005.

[33] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[34] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall.

Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[35] Sebastian Gerlach. *Fault-tolerant Dynamic Parallel Schedules.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, 2006.

[36] Sebastian Gerlach and Roger D. Hersch. Fault-tolerant parallel applications with Dynamic Parallel Schedules. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 8, April 2005.

[37] Sebastian Gerlach and Basile Schaeli. Dynamic parallel schedules. `http://dps.epfl.ch`.

[38] Sebastian Gerlach, Basile Schaeli, and Roger D. Hersch. Fault-Tolerant Parallel Applications with Dynamic Parallel Schedules: A Programmer's Perspective. In *Dependable Systems: Software, Computing, Networks*, volume 4028 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2006.

[39] L. Glimcher and Gagan Agrawal. A Performance Prediction Framework for Grid-Based Data Mining Applications. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[40] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*, pages 94–116. The Johns Hopkins University Press, 1996.

[41] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference (Vol. 2).* MIT Press, 1998.

[42] Ralf Gruber, Vincent Keller, Pierre Kuonen, Marie-Christine Sawley, Basile Schaeli, Ali Tolou, Marc Torruella, and Trach-Minh Tran. Towards an Intelligent Grid Scheduling System. In *Parallel Processing and Applied Mathematics*, volume 3911 of *Lecture Notes in Computer Science*, pages 751–757. Springer, 2006.

[43] Ralf Gruber, Pieter Volgers, Alessandro De Vita, Massimiliano Stengel, and Trach-Minh Tran. Parameterisation to tailor commodity clusters to applications. *Future Generation Computer Systems*, 19:111–120, 2003.

[44] Y.-K. Jun H.-D. Park. Detecting the first races in parallel programs with ordered synchronization. In *ICPADS '98: Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, pages 201–208, Washington, DC, USA, 1998. IEEE Computer Society.

[45] W. Haque. Concurrent deadlock detection in parallel programs. *Int. J. Comput. Appl.*, 28(1):19–25, 2006.

[46] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[47] B. Hong and V. K. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 52+, 2004.

[48] Chul-Eui Hong, Bum-Sik Lee, Gi-Won On, and Dong-Hae Chi. Replay for Debugging MPI Parallel Programs. In *MPIDC '96: Proceedings of the Second MPI Developers Conference*, page 156, Washington, DC, USA, 1996. IEEE Computer Society.

[49] Robert Hood. The p2d2 project: building a portable distributed debugger. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 127–136, New York, NY, USA, 1996. ACM.

[50] Paul Hsieh. Hash functions. `http://www.azillionmonkeys.com/qed/hash.html`.

[51] Daniel Jackson. Automating first-order relational logic. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139, New York, NY, USA, 2000. ACM.

[52] IBM journal of Research and Development staff. Overview of the ibm blue gene/p project. *IBM J. Res. Dev.*, 52(1):199–220, 2008.

[53] Rashmi Jyothi, O.S. Lawlor, and L.V. Kale. Debugging support for charm++. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 264–, April 2004.

[54] L. V. Kale, S. Kumar, and J. Desouza. A Malleable-Job System for Timeshared Parallel Machines. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 230–230, 2002.

[55] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[56] Prabhanjan Kambadur, Douglas Gregor, Andrew Lumsdaine, and Amey Dharurkar. Modernizing the C++ Interface to MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 266–274. Springer, 2006.

[57] Vincent Keller. *Optimal application-oriented resource brokering in a high performance computing grid*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, 2008.

[58] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 39–39, 2001.

[59] R. Kilgore and C. Chase. Re-execution of distributed programs to detect bugs hidden by racing. In *HICSS '97: Proceedings of the 30th Hawaii International Conference on System Sciences*, page 423, Washington, DC, USA, 1997. IEEE Computer Society.

[60] Eileen Kraemer and John T. Stasko. The visualization of parallel systems: an overview. *J. Parallel Distrib. Comput.*, 18(2):105–117, 1993.

[61] Dieter Kranzlmüller. Scalable parallel program debugging with process isolation and grouping. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 294, Washington, DC, USA, 2002. IEEE Computer Society.

[62] Jakub Kurzak and Jack Dongarra. Implementation of Mixed Precision in Solving Systems of Linear Equations on the Cell Processor. *Concurrency and Computation: Practice and Experience*, 19:1371–1385, 2007.

[63] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[64] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[65] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[66] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[67] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program executions using multiple views. *J. Parallel Distrib. Comput.*, 9(2):203–217, 1990.

[68] Lie-Quan Lee and Andrew Lumsdaine. The Generic Message Passing Framework. In *IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2003. IEEE Computer Society.

[69] De-Ron Liang and S. K. Tripathi. On performance prediction of parallel computations with precedent constraints. *Parallel and Distributed Systems, IEEE Transactions on*, 11(5):491–508, 2000.

[70] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988. `http://www.cs.wisc.edu/condor/`.

[71] J. Liu, Balasubramanian Chandrasekaran, W. Yu, J. Wu, D. Buntinas, Sushmitha Kini, D.K. Panda, and P. Wyckoff. Microbenchmark performance comparison of high-speed cluster interconnects. *Micro, IEEE*, 24(1):42–51, Jan.-Feb. 2004.

[72] Jiuxing Liu, B. Chandrasekaran, Jiesheng Wu, Weihang Jiang, S. Kini, Weikuan Yu, D. Buntinas, P. Wyckoff, and D.K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. *Supercomputing, 2003 ACM/IEEE Conference*, pages 58–58, Nov. 2003.

[73] Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[74] Steven S. Lumetta and David E. Culler. The mantis parallel debugger. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 118–126, New York, NY, USA, 1996. ACM.

[75] Svetlin A. Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008.

[76] M. Matsuda, T. Kudoh, and Y. Ishikawa. Evaluation of MPI implementations on grid-connected clusters using an emulated WAN environment. *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 10–17, May 2003.

[77] Message Passing Interface Forum. `http://www.mpi-forum.org`.

[78] Neeraj Mittal and Vijay K. Garg. Debugging distributed programs using controlled re-execution. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 239–248, New York, NY, USA, 2000. ACM.

[79] S. Morin, I. Koren, and C.M. Krishna. JMPI: implementing the message passing standard in Java. *IPDPS 2002: Proc. of the International Parallel and Distributed Processing Symposium*, pages 118–123, 2002.

[80] Mohamed Mosbah and Rodrigue Ossamy. Checking global properties for local computations in graphs with applications to invariant testing. In *ENC '04: Proceedings of the Fifth Mexican International Conference in Computer Science*, pages 35–42, Washington, DC, USA, 2004. IEEE Computer Society.

[81] Robert H. B. Netzer, Timothy W. Brennan, and Suresh K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 31–40, New York, NY, USA, 1996. ACM.

[82] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996.

[83] NVidia. Applications, Academic Papers and Articles Using CUDA. `http://www.nvidia.com/object/cuda_showcase.html`.

[84] M. Otta and S. Racek. A method for testing and debugging distributed applications. *EUROCON'2001, Trends in Communications, International Conference on.*, 2:548–551 vol.2, 2001.

[85] G. D. Peterson and R. D. Chamberlain. Stealing cycles: Can we get along? In *System Sciences, 1995. Vol. II. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 422–431 vol.2, 1995.

[86] S. Prakash and R. L. Bagrodia. MPI-SIM: using parallel simulation to evaluate MPI programs. In *Simulation Conference Proceedings, 1998. Winter*, volume 1, pages 467–474 vol.1, 1998.

[87] Sundeep Prakash. *Performance Prediction of Parallel Programs*. PhD thesis, University of California, Los Angeles (UCLA), Los Angeles, 1996.

[88] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24, New York, NY, USA, 2004. ACM.

[89] Hassan Rasheed, Ralf Gruber, Vincent Keller, Wolfgang Ziegler, Oliver Waeldrich, Philipp Wieder, and Pierre Kuonen. IANOS: An Intelligent Application Oriented Scheduling Framework For An HPCN Grid. In *Grid Computing*, Lecture Notes in Computer Science, pages 237–248. Springer, 2008.

[90] M.J. Rashti and A. Afsahi. 10-gigabit iwarp ethernet: Comparative performance analysis with infiniband and myrinet-10g. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[91] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. Skampi: a comprehensive benchmark for public benchmarking of mpi. *Sci. Program.*, 10(1):55–65, 2002.

[92] J.F. Ruscio, M.A. Heffner, and Srinidhi Varadarajan. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[93] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76, New York, NY, USA, 2005. ACM.

[94] S. Sankaran, J.M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *Int'l Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[95] Scalable LAPACK (Linear Algebra PACKage). `http://www.netlib.org/scalapack/`,.

[96] B. Schaeli, B. Gerlach, and R.D. Hersch. A simulator for parallel applications with dynamically varying compute node allocation. *IPDPS'06: Proceedings of the 20th International Parallel and Distributed Processing Symposium, Int'l Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS)*, pages 8 pp.–, April 2006.

[97] B. Schaeli, S. Gerlach, and R.D. Hersch. Decomposing partial order execution graphs to improve message race detection. *IPDPS'07: Proceedings of the 21st International Parallel and Distributed Processing Symposium, 12th Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-ToPMoDRS)*, pages 1–8, March 2007.

[98] Basile Schaeli, Ali Al-Shabibi, and Roger D. Hersch. Visual debugging of MPI applications. In *EuroPVM/MPI '08: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 239–247. Springer, 2008.

[99] Basile Schaeli and Sebastian Gerlach. The autoserial library. `http://home.gna.org/autoserial/`.

[100] Basile Schaeli, Sebastian Gerlach, and Roger D. Hersch. A simulator for adaptive parallel applications. *J. Comput. System Sci.*, 74(6):983–999, 2008.

[101] Basile Schaeli and Roger D. Hersch. Dynamic testing of flow graph based parallel applications. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis and debugging*, pages 1–10, New York, NY, USA, 2008. ACM.

[102] S. Sharma, Chung-Hsing Hsu, and Wu chun Feng. Making a case for a Green500 list. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

[103] Subodh Sharma, Sarvani Vakkalanka, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. A formal approach to detect functionally irrelevant barriers in mpi programs. In *EuroPVM/MPI '08: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 265–273. Springer, 2008.

[104] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Process scheduling challenges in the era of multi-core processors. *Intel Technology Journal*, 11, 2007.

[105] Stephen F. Siegel. Verifying parallel programs with mpi-spin. In *EuroPVM/MPI '07: Proceedings of the 14th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer, 2007.

[106] Stephen F. Siegel and George S. Avrunin. Verification of MPI-based software for scientific computation. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 286–303. Springer-Verlag, 2004.

[107] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free mpi programs for verification. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 95–106, New York, NY, USA, 2005. ACM.

[108] Stephen F. Siegel and George S. Avrunin. Verification of halting properties for mpi programs using nonblocking operations. In *EuroPVM/MPI '07: Proceedings of the 14th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 326–334. Springer, 2007.

[109] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[110] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference (Vol. 1), 2nd edition.* MIT Press, 1998.

[111] M.J. Sottile, V.P. Chandu, and D.A. Bader. Performance analysis of parallel programs via message-passing graph traversal. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[112] V. S. Sunderam. Pvm: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990. `http://www.csm.ornl.gov/pvm/`.

[113] Andrew Tanenbaum. *Modern Operating Systems*, pages 81–100. Prentice Hall, 2001.

[114] TotalView Technologies. the TotalView Debugger. `http://www.totalviewtech.com`.

[115] Rajeev Thakur, William Gropp, and Ewing Lusk. Mpich2. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[116] Nam Thoai, Dieter Kranzlmüller, and Jens Volkert. Shortcut replay: A replay technique for debugging long-running parallel programs. In *ASIAN '02: Proceedings of the 7th Asian Computing Science Conference on Advances in Computing Science*, pages 34–46, London, UK, 2002. Springer-Verlag.

[117] G. Utrera, J. Corbalan, and J. Labarta. Implementing malleability on MPI jobs. In *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, pages 215–224, 2004.

[118] Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Scheduling considerations for building dynamic verification tools for mpi. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis and debugging*, pages 1–6, New York, NY, USA, 2008. ACM.

[119] Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Implementing efficient dynamic formal verification methods for mpi programs. In *EuroPVM/MPI '08: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2008.

[120] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In *CAV '0: Proceedings of the 20th International Conference on Computer Aided*

Verification, volume 5123 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2008.

[121] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. Isp: a tool for model checking mpi programs. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 285–286, New York, NY, USA, 2008. ACM.

[122] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 51, Washington, DC, USA, 2000. IEEE Computer Society.

[123] Abhinav Vishnu, B. Benton, and D.K. Panda. High Performance MPI on IBM 12x InfiniBand Architecture. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[124] Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis de Supinski, and Andreas Sæbjørnsen. Improving distributed memory applications testing by message perturbation. In *PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, 2006.

[125] L. Wang and S.D. Stoller. Runtime analysis of atomicity for multithreaded programs. *Software Engineering, IEEE Transactions on*, 32(2):93–110, Feb. 2006.

[126] Jeremiah Willcock, Andrew Lumsdaine, and Arch Robison. Using MPI with C# and the Common Language Infrastructure. *Concurrency and Computation: Practice and Experience*, 17(7-8):895–917, 2005.

[127] Min-You Wu, Wei Shu, and Jun Gu. Efficient local search far DAG scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):617–627, Jun 2001.

[128] Qilong Zheng, Guoliang Chen, and Liusheng Huang. Optimal Record and Replay for Debugging of Nondeterministic MPI/PVM Programs. *Proceedings of the 4th International Conference on High Performance Computing in the Asia-Pacific Region*, 01:473–475, 2000.

List of Figures

List of Tables

Listings

Curriculum Vitæ

Basile Schaeli was born in 1979 in Lausanne. During his studies in Communication Systems at EPFL, he spent a year at KTH in Stockholm, Sweden, as an Erasmus student. From April to October 2001, he worked as a research intern at the IBM T.J. Watson Research Center in Hawthorne, New York. He then carried out his Master's thesis at the Indian Institute of Technology of Delhi (IITD), India.

After graduation, he joined the Peripheral Systems Laboratory in September 2003 to pursue a PhD thesis under the supervision of Professor Roger D. Hersch. He was a teaching assistant for Bachelor level Java programming course ("Programmation I & II"). He was then the lead teaching assistant and part-time lecturer for the Bachelor level "Programmation III" C/C++ programming course and for the Master level course "Parallel programming on clusters of workstations". He received two awards from the EPFL School of Computer and Communication Sciences for his teaching and science popularization contributions.

During his free time, he is an active member of the gymnastics club Vevey-Ancienne. Among other things, he is in charge of its bimonthly newsletter, and he co-authors stories and acts during the club's annual exhibition.

Personal bibliography

[42] R. Gruber, V. Keller, P. Kuonen, M.-Ch. Sawley, B. Schaeli, A. Tolou, M. Torruella and T.-M. Tran, "Towards an Intelligent GRID Scheduling System", *Proc. of Conference on Parallel Processing and Applied Mathematics (PPAM) 2005*, Poznan, Poland, Lecture Notes in Computer Science (LNCS) vol. 3911, pp. 751-757, Springer Verlag, 2006

[5] F. Armougom, S. Moretti, O. Poirot, S. Audic, P. Dumas, B. Schaeli, V. Keduas and C. Notredame, "Expresso: Automatic incorporation of structural information in multiple sequence alignments using 3D-Coffee", Nucleic Acids Research, vol. 34, pp. 604-608, 2006

[38] S. Gerlach, B. Schaeli, R.D. Hersch, “Fault-tolerant Parallel Applications with Dynamic Parallel Schedules: A programmer’s perspective”, J. Kohlas, B. Meyer and A. Schiper (Eds.): *Dependable Systems*, Lecture Notes in Computer Science (LNCS) vol. 4028, pp. 195-210, Springer Verlag, 2006

[96] B. Schaeli, S. Gerlach and R.D. Hersch, “A simulator for parallel applications with dynamically varying compute node allocation”, *Proc. IEEE IPDPS’06, 5th Int’l Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS)*, Rhodos, Greece, April 2006

[97] B. Schaeli, S. Gerlach and R.D. Hersch, “Decomposing Partial Order Execution Graphs to Improve Message Race Detection”, *Proc. IEEE IPDPS’07, 12th Int’l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-ToPMoDRS)* Long Beach, California, March 2007

[100] B. Schaeli, S. Gerlach and R.D. Hersch, “A simulator for adaptive parallel applications”, *J. Comput. System Sci.*, Vol. 74, Issue 6, pp. 983–999, Elsevier, , September 2008, `http://dx.doi.org/10.1016/j.jcss.2007.07.003`

[3] A. Al-Shabibi, S. Gerlach, R.D. Hersch and B Schaeli, “A debugger for flow graph based parallel applications”, *PADTAD ’07: Proc. 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pp. 14–20, London, UK, July 2007, `http://dx.dio.org/10.1145/1273647.1273651`

[101] B. Schaeli and R.D. Hersch, “Dynamic testing of flow graph based parallel applications”, *PADTAD’08: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis and debugging*, pages 1–10, Seattle, WA, July 2008. ACM.

[98] B. Schaeli, A. Al-Shabibi and R.D. Hersch, “Visual debugging of MPI applications”, *EuroPVM/MPI’08: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of Lecture Notes in Computer Science, pp. 239–247, Springer, 2008.