FAULT-TOLERANT DYNAMIC PARALLEL SCHEDULES

THÈSE N^{O} 3471 (2006)

PRÉSENTÉE LE 10 MARS 2006

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Laboratoire de systèmes périphériques

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Sebastian GERLACH

ingénieur en microtechnique diplômé EPF
et de nationalité allemande

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2006

Acknowledgements

First and foremost, I would like to thank my research director, Roger Hersch, for providing me the opportunity to work in his lab. The work involved not only the research and development leading to this thesis, but a wide array of other activities and research. I am particularly happy to have had the possibility to be lecturer for the peripherals course, and also to have worked on the lab's fantastic Visible Human projects.

Next I would like to thank all my colleagues, those who were in the lab when I started: Jean-Christophe Bessaud, Marc Mazzariol, and Patrick Emmel, those who came and went: Edouard Forler, Fabien Collaud, and Emzar Panikashvili, those who are still here: Emin Gabrielyan, Sylvain Chosson, Itzhak Amidror, and Basile Schaeli and also very particularly Fabienne Allaire. All these people contributed to make the lab an entertaining place to be.

I would also like to thank all the students who have worked on projects related to Dynamic Parallel Schedules (DPS). In chronological order: Olivier Bruchez, Simon Chatelain, and Dany Lauener, who worked on a parallel video processing application for editing high resolution uncompressed video files; Stéphane Magnenat and Luc-Olivier de Charrière, who implemented a parallel virtual biotope; Leonardo Peparolo and Lorenzo Cantelli, who developed a prototype Java binding for DPS; Nicolas Frey, who worked on parallel implementations of genetic algorithms; Pascal Jermini, who developed the Java version of the DPS trace analyzer tool; Pierre Dumas, who implemented a parallel version of the T-Coffee multiple sequence alignment application with DPS. Thanks also to all the students who took the parallel programming course at EPFL – suffering through the first versions of DPS, followed by smoother sailing in the later years.

Of course, I wouldn't be here if not for my family. I am therefore very grateful to my parents, Silke and Dieter Gerlach, for their unyielding support.

Greetings go out to all those I know in the demoscene, in particular you Calodox guys! Start doing stuff again!

Summary

Dynamic Parallel Schedules (DPS) is a high-level framework for developing parallel applications on distributed memory computers such as clusters of PCs. DPS applications are defined by using directed acyclic flow graphs composed of user-defined operations. These operations derive from basic concepts provided by the framework: split, merge, leaf and stream operations. Whereas a simple parallel application can be expressed with a split-leaf-merge sequence of operations, flow graphs of arbitrary complexity can be created. DPS provides run-time support for dynamically mapping flow graph operations onto the nodes of a cluster.

The flow graph based application description used in DPS allows the framework to offer many additional features, most of these transparently to the application developer. In order to maximize performance, DPS applications benefit from automatic overlapping of computations and communications and from implicit pipelining. The framework provides simple primitives for flow control and load balancing. Applications can integrate flow graph parts provided by other applications as parallel components.

Since the mapping of DPS applications to processing nodes can be dynamically changed at runtime, DPS provides a basis for developing malleable applications. The DPS framework provides a complete fault tolerance mechanism based on the dynamic mapping capabilities, ensuring continued execution of parallel applications even in the presence of multiple node failures.

DPS is provided as an open-source, cross-platform C++ library allowing DPS applications and services to run on heterogeneous clusters.

Keywords: parallel programming, fault-tolerance, dynamic resource allocation, parallel application frameworks, flow graphs, parallel schedules

Résumé

Dynamic Parallel Schedules (DPS) est un framework pour le développement d'applications parallèles fournissant un haut niveau d'abstraction. DPS est conçu pour des machines à mémoire distribuée, telles que des clusters de PCs. Une application DPS est décrite au moyen d'un graphe de flux acyclique orienté composé d'opérations définies par le développeur. Ces opérations fonctionnent selon des concepts de base fournis par le framework: *split*, *merge*, *leaf*, et *stream*. Bien qu'une application simple peut être décrite au moyen d'une séquence split-leaf-merge, des graphes de flux d'une complexité arbitraire peuvent être créés. DPS déploie dynamiquement les opérations du graphe de flux sur les nœuds de calcul du cluster durant l'exécution de l'application.

La description d'applications basée sur un graphe de flux utilisée dans DPS permet au framework d'offrir de nombreuses fonctionnalités complémentaires, qui ne requièrent pour la plupart aucune intervention du programmeur. Afin de maximiser la performance, les applications DPS bénéficient d'une superposition automatique des calculs et des communications et d'un pipelining implicite des opérations. Le framework fournit des primitives simples pour le contrôle de flux et l'équilibrage des charges. Des applications peuvent intégrer des parties de graphes de flux exposées par d'autres applications comme des composants parallèles.

Dans la mesure où le déploiement d'une application DPS vers les nœuds de calcul peut être modifié dynamiquement en cours d'exécution, DPS fournit une base pour le développement d'applications malléables. Le framework DPS fournit également un mécanisme complet de tolérance aux pannes basé sur ces capacités de déploiement dynamiques, assurant ainsi une exécution en continu d'une application parallèle même en présence de pannes multiples.

DPS est fourni sous forme d'une librairie C++ multi-plateformes permettant aux applications et aux services DPS de fonctionner sur des clusters hétérogènes. DPS est distribué sous la licence libre GPL.

Mots clé: programmation parallèle, tolérance aux pannes, allocation dynamique de ressources, frameworks de développement d'applications parallèles, graphes de flux, parallel schedules

Table of Contents

Chapter 1

Introduction

Parallel computation has always been an attractive solution whenever problems with very high computation requirements need to be solved. The concept of splitting complex tasks into many smaller tasks and executing these smaller tasks in parallel on multiple processors is very intuitive. Nevertheless, implementing efficient parallel programs remains a daunting task. Therefore, using high-level programming languages has the potential of simplifying the development of parallel applications. This chapter presents an overview of various existing approaches to parallel programming.

1.1 Introduction

The use of parallel computing systems for accelerating complex computations is not a recent phenomenon. Early examples of parallel machines can be traced back to the late 1950's, with the Burroughs D-825 [Enslow77][Gray03] and the American National Bureau of Standards PILOT [Leiner59]. The Burroughs D-825 is a *multiprocessor* system, where multiple processing units are interconnected with a single, shared memory. The PILOT is a *multicomputer*, where multiple computers are connected together with a communication network. Both of these architectures are still in widespread use nowadays. The systems currently at the top of the Top500 supercomputer list[1] are representatives of these two architectures: the IBM BlueGene/L [Adiga02] is a multicomputer system, whereas the SGI Altix [Brooks05] is a multiprocessor system. These two systems are finely tuned and designed for maximum performance and represent a large investment in hardware design.

Figure 1. The Burroughs D-825 (photograph by Burroughs Corporation)

Both types of architecture provide the ability to simultaneously execute multiple operation streams on independent processors. On multiprocessors, the individual execution streams cooperate by reading and writing to the shared memory. On multicomputers, the execution streams cooperate by sending and receiving messages over the communication network. Modern parallel systems often use a combination of both architectures, where multiprocessor nodes usually containing 2 or 4 processors are interconnected to the other nodes by a high-performance network. These types of systems are called *clusters*. In order to

[1] At the time of writing, the current Top500 list is the one released in June 2005 at ISC2005. The latest list can be found at http://www.top500.org

achieve maximum efficiency, applications running on clusters need to use both the shared-memory and message-passing communication paradigms [Baden00].

Clusters are currently enjoying a very high popularity within the high performance computing community. Most clusters interconnect large numbers of commodity off-the-shelf components with a high-performance network. Currently 60% of the systems listed on the Top500 list are of this type. These systems are usually inexpensive to build, depending on the workstations and network used, but they do have many shortcomings. They are not efficient in their use of space, their power consumption, and their heat dissipation, since the individual component workstations have not been optimized for this type of use. They are also less reliable, since there are many individual components that can fail in every system.

On a smaller scale, current high-performance processor designs are moving away from ever higher clock rates and complex architectures to the integration of multiple processing cores within a single chip. Therefore, in the near future all high performance computing devices, from workstations to game consoles, will be multiprocessor parallel systems.

With these latest developments in parallel computing hardware, parallel computing is gaining ever more presence, and moves into the mainstream. Efficient and inexpensive parallel hardware is nowadays available. However, the development of software for these machines still remains a major hurdle for most developers and scientists. Producing highly efficient software that is finely tuned for one specific parallel computer remains a task for specialists.

There have been many proposals for libraries, languages and models designed to ease the parallel application developer's burden. This dissertation presents one such approach in the form of *Dynamic Parallel Schedules* (DPS). The presentation of DPS begins in Chapter 2, with a general overview of the parallel schedule approach. The remainder of the present chapter focuses on fundamental aspects of parallel programming, and also illustrates some other approaches to parallel programming libraries and languages.

1.2 Parallel programming challenges

When developing software for single-processor machines, application developers focus on the algorithms they are implementing. The developer needs to consider aspects such as the algorithms' computational requirements and memory footprint in order to produce efficient programs. When moving to a parallel architecture, multiple new challenges arise for the developer when implementing an algorithm: distribution of tasks onto multiple execution threads, data distribution, and the collaboration between multiple execution threads. Additional challenges concern runtime aspects, such as optimizing resource usage and debugging. These challenges are now briefly presented.

1.2.1 Task distribution

On a parallel architecture, the application gains the ability to execute multiple instruction flows simultaneously. Therefore, the developer must partition the computations performed by the algorithm into multiple groups that will be dispatched to the available processors. For instance, consider a simple bitmap image processing operation, where a filter is applied for example to blur the image. The blur operation is performed by applying a small fixed-size convolution matrix to the whole image. Separating this operation is very simple, since there are no dependencies between pixels, and therefore every single pixel can potentially be handled by a different processor. Let us consider another bitmap image processing

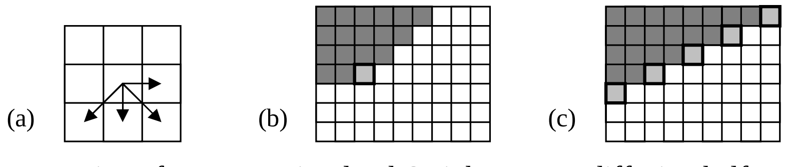


Figure 2. (a) Propagation of error term in Floyd-Steinberg error diffusion halftoning, (b) resulting dependencies for a pixel in the image, (c) pixels that can be computed in parallel, provided all the dark gray pixels have already been computed.

operation: Floyd-Steinberg error diffusion halftoning [Metaxas99]. When applying this algorithm, an error term is propagated over the image, leading to numerous pixel dependencies illustrated in Figure 2. These dependencies limit the number of pixels that can be computed simultaneously.

1.2.2 Data distribution

When tasks are distributed onto multiple processors, the processors need to access the data on which to perform their processing operations. For instance, when performing an image blur, the processor performing the operation on one pixel of the image needs to access that pixel, and also a limited-size neighborhood of that pixel. Therefore, if the application is running on a distributed memory system with *n* processors, a possible distribution is to provide each processor with one *n*th of the image (for example a horizontal band) and additional neighborhood pixels on the borders.

Such a simple data distribution scheme is only possible for simple applications. Consider a matrix multiplication where two different square matrices are multiplied one with another. The matrix multiplication is performed by computing the dot product of every row of the first matrix by every column of the second matrix. An initial data distribution for the first matrix may consist of giving each participating processor a portion of its rows, evenly distributing the complete matrix. However, each processor still needs the complete second matrix in order to be able to perform its required computations. No trivial symmetric distribution is possible as in the image processing example.

The distribution of data is of particular importance on multicomputer systems, since one processor cannot transparently access data stored in another processor's address space. Any transfer of data requires explicit use of the network interconnecting the computers, usually incurring high latencies. Even on multiprocessor systems, due to the system's architecture, there is a varying cost associated with accessing memory depending on where it is located.

1.2.3 Collaboration

Separate processing threads executing a parallel application cannot proceed in an uncoordinated fashion. Even for the simple image processing application, a minimal amount of collaboration is required: the execution threads must gain access to the data they have to process, execute their task, and finally the result must be gathered when processing is complete. The manner in which this collaboration is performed is highly dependent on the system architecture.

For a shared memory system, the process is fairly straightforward. Initially the image is loaded into memory taking into account the locality of the memory relative to the processors that will be used later. Subsequently, a number of processing threads are spawned to execute the operation on multiple processors. The thread that started the application waits for the termination of the threads it spawned, and can save the processed image from memory. The standard instruments for implementing collaboration on this type of system are operating-system level primitives (mutexes, conditions, semaphores, etc.)

For distributed memory systems, all collaboration tasks are accomplished by sending messages over the network. Using messages involves additional overheads due to the additional software and hardware network layers, as well as the additional context information that needs to be transferred.

1.2.4 Optimal resource usage

The underlying hardware of processing nodes participating in a parallel computation is often capable of performing multiple tasks simultaneously. For example, in a multicomputer system, nodes can perform computations while transferring data over the network, both for transmission and reception purposes (full-duplex communication). By carefully analyzing the data and collaboration dependencies within an application, it is frequently possible to overlap a portion of the communications with computations in order to optimize the usage of the available resources.

1.2.5 Debugging

When working with multiple execution threads on one or more computers, debugging applications becomes a challenging task. Standard debuggers for sequential applications focus on the ability to step from one instruction to the next, and to analyze where errors are taking place. While this type of debugger can still be used in parallel applications by single-stepping through individual threads of execution, there are many types of errors that cannot be uncovered by this process. Parallel applications are highly sensitive to the timing of their individual tasks, exhibiting behaviors such as race conditions or deadlocks. These time-dependent issues are usually very difficult to reproduce reliably within a debugging environment. There are many more possible sources for errors, since flaws in the program can also reside within inconsistent data distributions or erroneous collaboration between tasks.

1.2.6 Fault tolerance

Parallel applications rely on large computing systems composed of many individual elements for their execution. While single components may function reliably for several years, this is no longer the case for parallel systems containing hundreds or thousands of components. The likelihood of a component failure during a longer program execution is very high. Component failures can be handled at the hardware level, for example by providing redundant components, but this usually leads to very high costs. Therefore the responsibility of handling failures is often left to the software running on parallel systems.

1.3 Parallel programming models

To meet the challenges posed by parallel programming, several approaches to application development have emerged. The most attractive solution is a parallelizing compiler that can process normal sequential code and generate a corresponding parallel program. Another solution is to use a programming language that is designed specifically for parallel computation. Finally, a conventional sequential language and compiler can be used together with a parallel application library that provides parallelization primitives or services. These three models will now be discussed in more detail. A more detailed overview of parallel programming models and languages can be found in [Skillicorn98].

1.3.1 Parallelizing compilers

Parallelizing compilers represent the most user-friendly solution to parallel application development. Software developers write their programs in a familiar sequential language, and leave all the parallelization tasks to the compiler. Ideally, the parallelizing compiler would thoroughly optimize the resulting code for a given parallel machine in the same manner a sequential compiler generates highly optimized binaries for specific target system architectures.

Creating such parallelizing compilers, though, is a difficult task, and the focus of many ongoing research projects. A parallelizing compiler needs to be able to automatically detect parallelism within a sequential implementation of an algorithm in a traditional language such as C or FORTRAN. Many techniques have been developed for exposing parallelism within sequential code [Bacon94][Gupta99]. One possible source of parallelism is to locate independent procedure calls, and to perform these calls simultaneously [Burke86]. Another possibility is to analyze loop structures within the application, and transform the loops in order to increase data locality [Wolf91][Wolfe95]. The transformation of the application in order to make use of the parallelism needs to ensure that the resulting implementation is efficient. It therefore needs to minimize data exchanges between the parallel tasks and to minimize synchronization requirements.

Examples of parallelizing compilers for traditional language include SUIF [Hall96] and dHPF [MellorCrummey01]. There are also specific implementations for particular parallel systems such as multiprocessor DSPs [Franke05].

1.3.2 Parallel languages

Due to the immense difficulty of developing efficient, fully automatic parallelizing compilers for existing programming languages, it is tempting to develop new programming languages that are designed specifically for developing parallel applications. Parallel languages can either be based on an existing language such as C or FORTRAN, or be designed from scratch. Basing a parallel language on an existing programming language has many advantages, since developers are usually already familiar with most of its concepts, and existing implementations provide a good starting point for developing new compilers. Elements that make code analysis difficult can be removed from the existing language, such as pointer arithmetic in C, and new elements that are specifically tailored for parallel applications can be added. Starting from scratch has the advantage of not keeping any legacy elements that might have a negative effect on performance or ease of implementation, but makes it necessary for potential users to learn a new set of language constructs.

Examples of languages that are very closely based on existing languages are Unified Parallel C (UPC) [ElGhazawi05] and OpenMP-C [OpenMP05], which are both based on C. These two languages provide extensions that allow developers to annotate loops that should be parallelized and provide hints about the parallelization. Another popular language based on C is Charm [Kale94], which uses a message-driven execution model. Other languages such as ParoC++ [Nguyen03] expand an existing object-oriented programming language (in this case C++) in order to support shared parallel objects. On the other side of the spectrum are languages such as ZPL [Chamberlain00] that use very high level application descriptions which do not resemble conventional programming languages. These high level descriptions are however easier to parallelize automatically than C for example.

Other solutions are based on skeletons, which provide pre-implemented parallel constructs. The programmer can combine and customize these constructs by providing his own code. Skeletons exist for task-parallel and data-parallel constructs [Kuchen02]. Task-parallel constructs are based on the transfer of data items between tasks. Worker tasks process data items as they arrive and send off processed data items as soon as they are ready. Data-parallel constructs work on a distributed data structure that is stored within the worker tasks. Internally, skeletons are usually represented as a tree with nested parallel constructs. Skeleton languages also allow creating sequences of constructs. Examples of languages based on skeletons are P3L [Bacci95] and Skil [Botorog96].

Other languages allow existing code segments to be combined using dependency graphs. These segments are subsequently scheduled for execution on the parallel system by taking into account their dependencies. Examples of graph based languages are the Network of Tasks model [Skillicorn00] and Mentat [Grimshaw93].

There are many more parallel language concepts and implementation examples; this short selection is by no means exhaustive.

1.3.3 Parallel libraries

Parallel libraries are currently the most popular means for writing parallel applications. Developing with a library is easy to get into, since programmers can keep the programming language and environment they are familiar with, and simply use the additional functionality provided by the library. Since the underlying programming language is sequential, it is always up to the programmer to explicitly decompose his application in order to expose small sequential blocks that can be executed in parallel. The manner in which these sequential blocks are subsequently used varies widely with the design of the library. Also, the handling of other tasks such as communication and synchronization can be entirely controlled by the library or left fully to the programmer.

The most popular libraries currently in use are MPI [Dongarra96] and PVM [Geist94]. These libraries provide low-level message-passing functions, but leave most of the other parallel programming issues to the programmer. As with parallel languages, many other libraries have been developed to simplify parallel application development beyond what is proposed by MPI and PVM. For example, there are C++ class libraries that implement skeletons [Kuchen02b].

1.4 DPS in the parallel programming landscape

The Dynamic Parallel Schedules framework which is the object of this thesis is a new framework for developing parallel applications. It is implemented as a library for the C++ programming language, in order to make it as easy to use as possible, and simplify integration into existing projects. The methodology used for developing parallel applications with DPS has some resemblance with task parallel skeletons, since simple constructs are provided that can be combined and extended by the programmer.

However, instead of combining the constructs in a tree by nesting them within one another, DPS applications are defined as directed acyclic graphs of constructs.

1.5 Document structure

The following chapters of this thesis are structured as follows: Chapter 2 presents the underlying concepts of parallel schedules in detail, and provides an example of how a parallel application can be created using these concepts. Chapter 3 presents the implementation of parallel schedules within a C++ library, and the features that the library provides in order to enable the development of dynamic, malleable parallel applications. Chapter 4 presents several applications that were developed using the DPS library, and analyzes how the implementation of parallel schedules influences their performance. Chapter 5 presents the internal structure of the DPS library and the execution model for executing parallel schedules. Chapter 6 presents the fault-tolerance mechanism provided by DPS and analyzes the performance overheads incurred by using fault-tolerance. Chapter 7 provides a conclusion and outlook on future research in respect to parallel schedules.

Beyond these chapters, three appendices are provided. Appendix A briefly presents a previous implementation of parallel schedules. Appendix B presents several tools that were developed in order to manage the execution of applications using DPS and analyze their performance in detail. Appendix C provides a deeper insight into the internal implementation of DPS, illustrating some of the C++ programming techniques used within the library.

Chapter 2

The Parallel Schedule Approach

The parallel schedule approach to parallel application programming relies on a high level model for describing parallel applications. Parallel schedules use directed acyclic flow graphs of customizable operations to describe applications. The operations are mapped onto threads running on multiple processing nodes. Parallel execution is achieved by using a flexible, compositional split – execute – merge paradigm. This chapter presents the concepts behind the parallel schedule approach.

2.1 Introduction

Parallel schedules are a model for describing and executing parallel applications. The application description is based on a data-driven graph of program flow which is mapped onto multiple processing nodes. The first part of this chapter presents the flow graph concept and how it can be used to describe an application. The second part illustrates how the components of the flow graph are mapped onto multiple processing nodes in order to achieve parallel execution. The third part gives an example of an application developed with parallel schedules. Finally, we present some of the runtime features available within parallel schedules.

DPS is an implementation of the concepts presented here in the form of a C++ library. The syntax used for expressing parallel schedules as well as some more advanced runtime capabilities are the object of Chapter 3.

2.2 Flow graphs

A parallel schedule describes a parallel application using a data-driven computational model. In order to illustrate this model, let us consider a simple sequential computation task, where some data is read from a file, followed by a processing operation. When expressed using simple sequential pseudocode, this computation task looks as follows:

```
readFile(inputFile, inputBuffer, inputBufferSize);
processData(inputBuffer, inputBufferSize,
            outputBuffer, outputBufferSize);
```

The first instruction reads data from a file to a newly allocated memory buffer, and the second instruction processes the data, storing the result in a second buffer. When converted to a data-driven model, the data elements appearing in the program drive its execution, as shown in Figure 3. The *ReadFileRequest* data object causes the execution of the *ReadFile* operation, which produces a *DataBuffer* output data object. The output data object of *ReadFile* is subsequently processed by *ProcessData*, which produces another *DataBuffer* output data object containing the results of the processing. The involved data objects contain the same information as the parameters passed to the functions in the previous pseudocode.

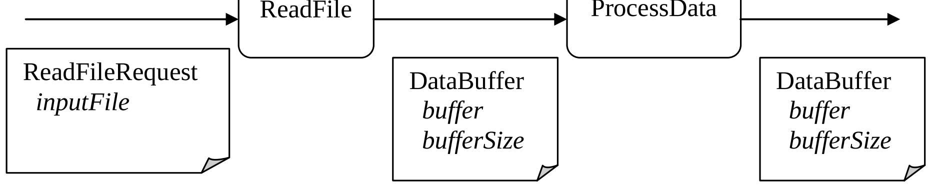


Figure 3. Simple data-driven program execution. The contents of the data objects are shown in italics.

The data-driven model relies on a graph of execution paths that can be followed by the data objects. The nodes on the graph are the operations that are executed, using a data object as input, and producing a new data object as output. Within the context of parallel schedules, we refer to such a graph as a *flow graph*.

2.2.1 Split-Process-Merge flow graphs

The previous example showed a simple sequential application, but the model can easily be extended to parallel applications by using special operations that can produce multiple output data objects or consume multiple input data objects. The flow graph in Figure 4 shows a simple parallel application.

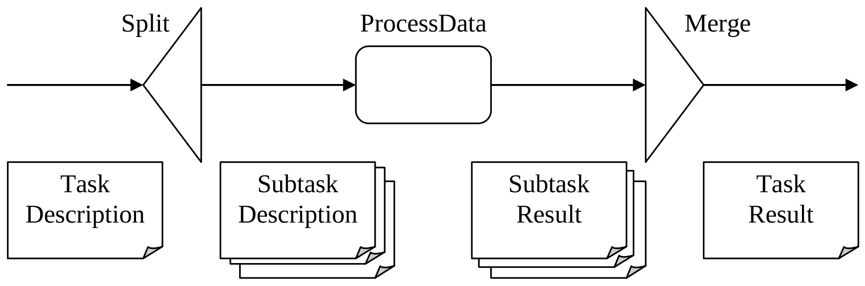


Figure 4. A simple parallel application using a Split-Process-Merge flow graph

The *Split* operation takes a single task description data object as input, and subdivides it into multiple subtask descriptions, each represented by one output data object. Each of these subtask description data objects is subsequently processed by the *ProcessData* operation, yielding a subtask result data object. All the subtask results are collected by the *Merge* operation, which combines all the subtask results into one task result. If the *ProcessData* operations are distributed onto multiple processing nodes, this flow graph corresponds to a simple compute farm application pattern. The mechanism used for allocating processing nodes for the operations is described later.

The simple *Split-Process-Merge* flow graph described above shows all the basic building blocks for parallel schedules. We can distinguish three basic operation types in this flow graph:

- Processing operations (also called *Leaf* operations) – these operations take exactly one data object as input, and produce exactly one data object as output.
- *Split* operations – these operations take exactly one data object as input, and can produce any number of data objects as output (but at least one, otherwise the data flow in the flow graph would be halted).
- *Merge* operations – these operations take any number of data objects as input, and produce exactly one data object as output. The number of input data objects corresponds to the number of data objects produced by the corresponding split operation.

All operations used within flow graphs can be fully customized by the application developer. In particular, both the split and merge operations can contain customized code to control exactly how data is distributed in subtasks, and how the computed sub-results are combined into the final output result. The data objects that drive the execution of the flow graph can also be freely defined by the developer.

2.2.2 Complex flow graphs

Operations can be combined to create flow graphs of arbitrary complexity. Three fundamental patterns can be distinguished in flow graph construction: pipelines, nested constructs and parallel branches. Additional constructs, such as stream operations and loops, can be used in order to construct complex pipelined flow graphs.

Pipelines are the simplest pattern, where two leaf operations follow each other. From the outside perspective, a sequence of two leaf operations is equivalent to a single leaf operation, since for each input data object, there is exactly one output data object. A basic pipelined sequence of leaf operations is shown in Figure 5. This equivalence property also applies to the simple split-process-merge flow graph, since for every data object that is input into the flow graph, exactly one data object is output, as illustrated in

Figure 6. The leaf operation within the split-process-merge sequence can thus be replaced by another split-process-merge graph, yielding a *nested* graph, as illustrated in Figure 7.

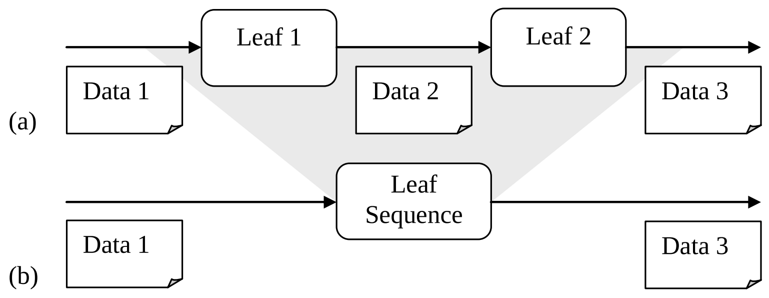


Figure 5. (a) Pipelined sequence of operations and (b) equivalent outside view of a pipelined sequence of operations

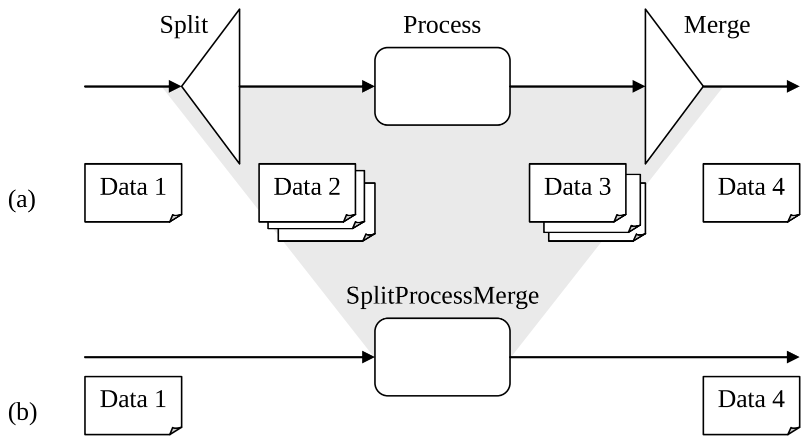


Figure 6. (a) Split-Process-Merge flow graph and (b) equivalent outside view of split-process-merge flow graph section

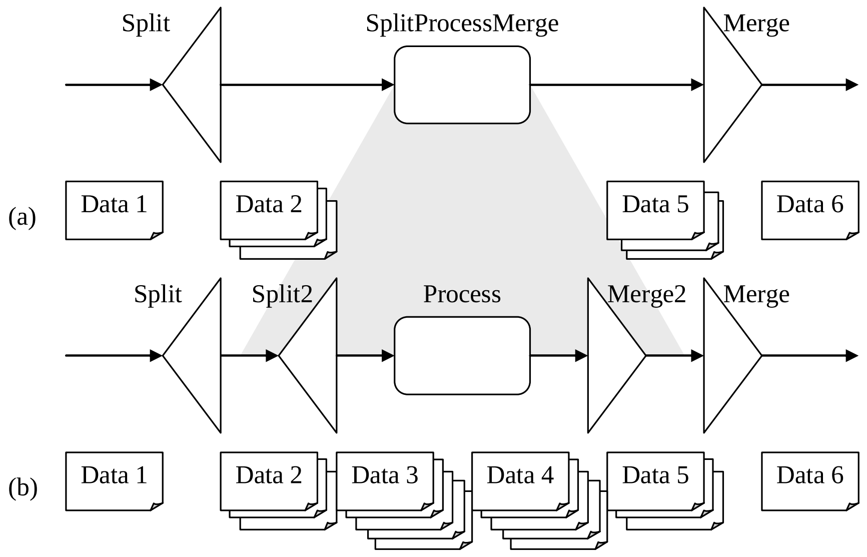


Figure 7. (a) Split-Process-Merge flow graph and (b) nested Split-Process-Merge flow graph

Since a split operation can create any number of output data objects, they should be able to be processed by different operations, yielding multiple *parallel branches* on the flow graph between the split operation and the merge operation.

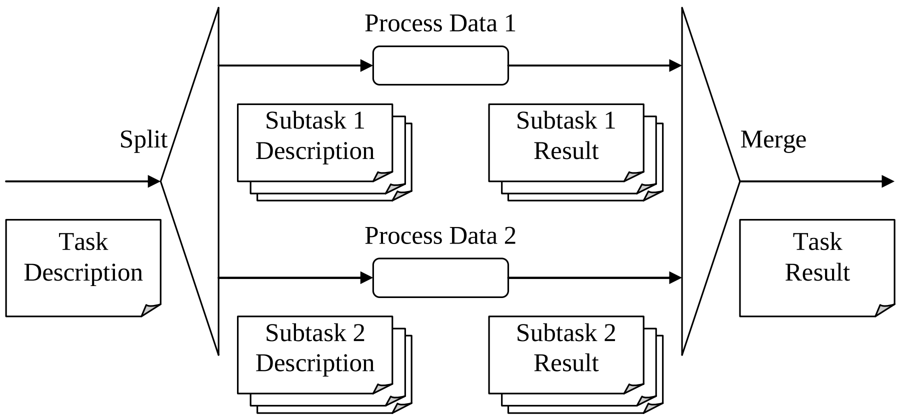


Figure 8. Parallel branches within a split-merge operation pair flow graph

2.2.3 Stream operations

The split-merge concept used in parallel schedules provides a very strict synchronization model. Applications often need to perform multiple computation steps, where the following computations need the results of previous computations in order to execute. When these computations are split up and performed in parallel, the synchronization usually is performed by ensuring that all the first computations have completed with a merge operation before splitting out the second computation as illustrated in Figure 9.

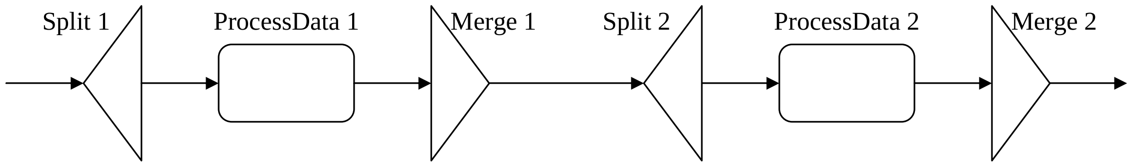


Figure 9. Two computations with intermediate synchronization

In some cases, not all the results of the parts of the first computation are required in order to perform parts of the second computation. 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, as illustrated in Figure 10. The stream operation can output data objects at any time within the merge process.

Stream operations can therefore be used for improving pipelining across synchronization points. An example illustrating the use of the stream operation and the improved pipelining that can be achieved is provided in Chapter 4, within the context of an LU decomposition application.

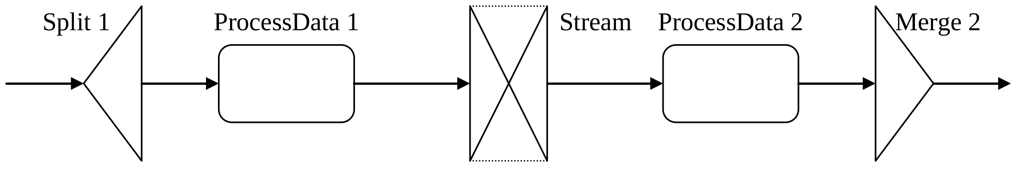


Figure 10. Two computations with streaming synchronization

2.2.4 Loops

For some applications, it might be necessary to repeatedly execute an operation on a data object until a certain condition is met. Within a flow graph, this type of execution pattern can be expressed as a *loop*. The loop is a specialized type of operation that encloses any sequence of operations and evaluates a condition on the output data object of this operation. As long as the condition is true, the encapsulated sequence of operations is executed again on the output data object. The encapsulated sequence of operations needs to have the same input and output data object types in order to ensure that looped execution is valid. 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.

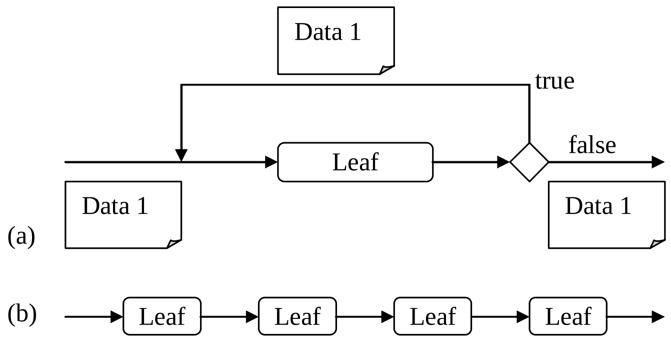


Figure 11. (a) Loop construct enclosing a leaf operation and (b) equivalent pipelined sequence of operations if the condition returned true on the first three iterations

Figure 12 illustrates some additional 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.

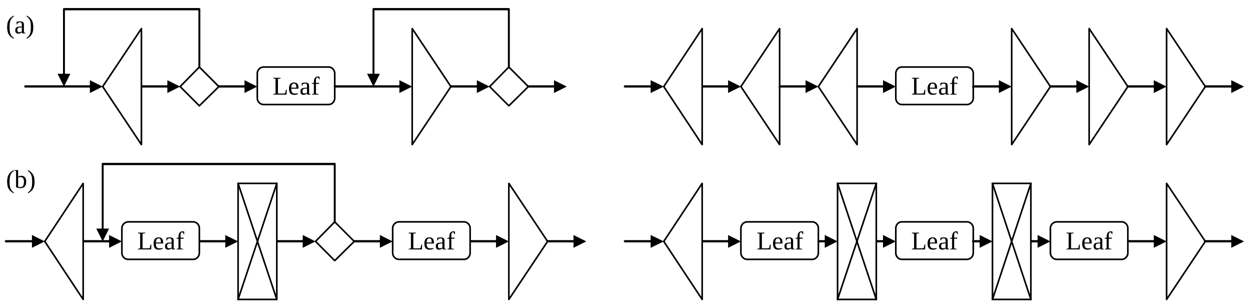


Figure 12. Other configurations using the loop construct (left) and resulting runtime flow graph (right). (a) multi-level split and merge operations, (b) deep pipeline using stream operations

By combining the operations and constructs shown in the previous sections, flow graphs of arbitrary complexity may be created. The flow graphs resulting from the basic patterns are always acyclic, since none of the patterns allows the construction of cycles. The acyclic property is particularly important for program execution, since it ensures that execution is always free of deadlocks.

2.3 Threads and thread collections

Flow graphs provide a simple and efficient mechanism for describing program flow. Since we want to use these flow graphs for parallel program execution, the description needs to be extended in order to indicate on which processing node the various operations on the flow graph should be executed. The processing node selection is performed by assigning operations 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, in the same way an operating system thread allows the execution of sequential code. Also like operating system threads, multiple parallel schedule threads can be located on the same processing node, and execute operations concurrently independently of each other. A parallel schedule thread typically executes only one operation at a time. In the following discussion, the word *thread* always refers to parallel schedule threads, unless the term *operating system thread* is explicitly specified.

The threads are grouped within *thread collections*, which contain multiple threads of the same type. The threads within a thread collection may be located on an arbitrary number of processing nodes.

To illustrate the parallel schedule threading model, let us 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, where the first is used for all the master tasks, and the second is used for all the processing tasks. The master thread collection contains only a single thread, whereas the processing thread collection contains one thread for each participating processing node.

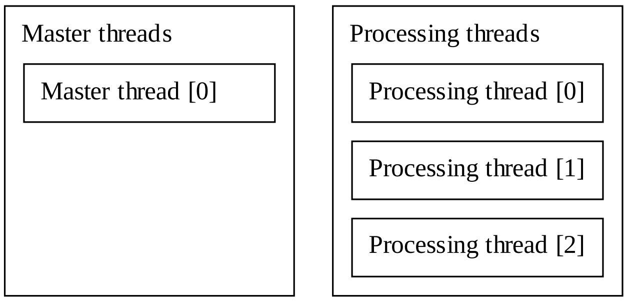


Figure 13. Example of thread collections for a simple compute farm application. Two thread collections, Master threads and Processing threads, are used.

In order to use the thread collections, they need to be attached to the operations within the flow graph. For each operation within the flow graph, one thread collection is chosen. In our compute farm example, the *Split* and *Merge* operations are attached to the Master threads collection, and the *ProcessData* operation is attached to the Processing threads collection.

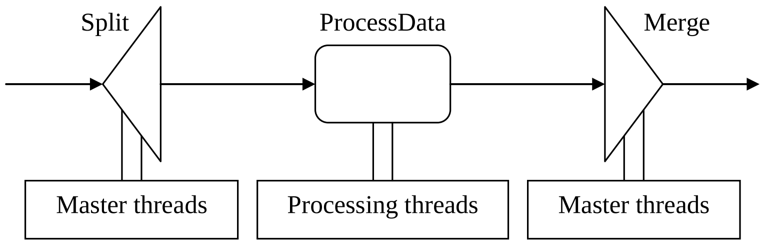


Figure 14. Assignment of thread collections to operations within the flow graph

This leaves one question unanswered during program execution: which thread within the collection should be used as the execution context for running an operation? To answer this question, an additional function is added to all operations within the flow graph: the *routing function*. The role of the routing function is to select a thread within the assigned thread collection. Since execution of operations is triggered by data objects in our data-driven computation model, the routing function can use the data object that will be processed as input by the operation as a parameter. The routing function is very flexible, since it can provide both simple mechanisms such as constant routing or round-robin routing, and complex mechanisms such as data-dependent routing (by using the data object) or even automatic load-balancing. For the simple compute farm example, the routing functions shown in Figure 15 can be used.

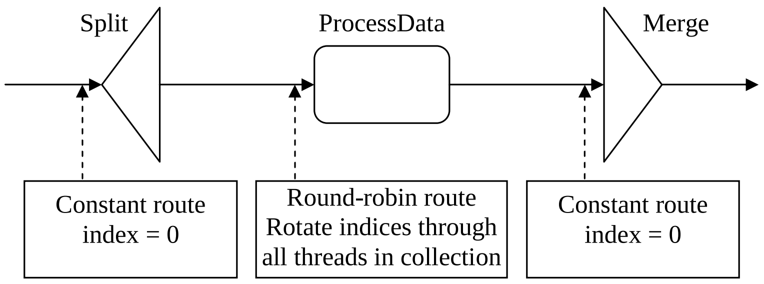


Figure 15. Routing functions for compute farm

The threads and thread collections provide a logical description of the execution environment of the application, since at this point the threads have not yet been assigned to processing nodes in the cluster. In order to complete the description, each thread of each collection must be mapped to the processing node where its assigned operations are to be executed. Multiple threads may be mapped to the same processing node. This assignment process is known as *thread mapping*.

2.3.1 Thread local storage

Just like operating system threads, 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. 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. The storage is specific to individual threads. Even when two threads of the same type are located on the same processing node, they cannot access each other's data.

2.4 Parallel schedules

The combination of a flow graph and the thread collections that are referenced by the flow graph forms a *parallel schedule*. The parallel schedule contains all the information that is required for executing the application. The previous section has shown a parallel schedule for a simple compute farm application. In this section we explore the parallel schedule for a non-trivial data-parallel application, a parallel implementation of a cellular automaton: Conway's game of life. The program structure of the game of life is very similar to many other problems, such as finite difference computations like thermal diffusion, or skeletonization in image processing.

The game of life was introduced in 1970 by the mathematician John Conway, and simulates the life of a colony of cells based on a set of simple rules. The cells are placed on a regular rectangular grid. Each square of the grid can either contain a living cell or a dead cell. The state of the cells on the grid evolves in cycles, where the state of the grid in the next cycle is computed based on the current state. When computing the state of a cell for the next cycle, its state depends on the current state of all its neighbors in an 8-neighborhood according to the following rules:

1. A living cell with 2 or 3 living neighbors stays alive, otherwise it dies.
2. A dead cell with exactly 3 living neighbors becomes alive, otherwise it stays dead.

The grid of cells wraps around, i.e. the right-hand neighbor of a cell in the last column is in the first column. The following figure shows a sample evolution on a 5x5 grid.

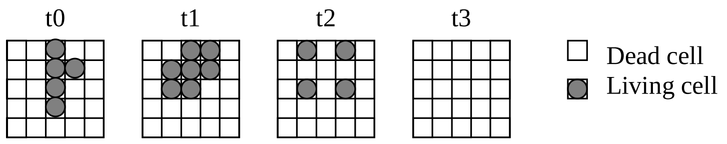


Figure 16. Evolution sequence of the game of life on a 5x5 grid

In the parallel schedule implementation of the game of life, the current state of the grid is distributed onto multiple threads, in order to enable parallel computation of the next state. The grid is divided into horizontal bands of equal size as illustrated in Figure 17. In order to compute the next state for their part of the grid, each thread needs in addition to the locally stored part of the grid the last line from the previous block and the first line from the next, so as to have a complete 8-neighborhood for each local cell. Therefore each thread has enough storage space for an additional two lines of the grid, which are used during computation in order to store the required upper and lower borders.

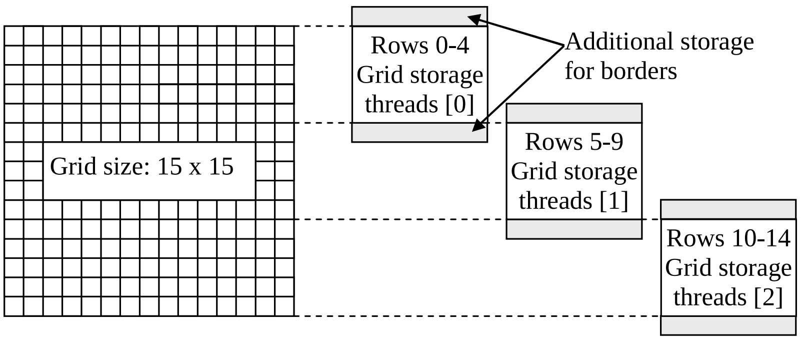

Grid size: 15 x 15
Rows 0-4 Grid storage threads [0]
Additional storage for borders
Rows 5-9 Grid storage threads [1]
Rows 10-14 Grid storage threads [2]

Figure 17. Distribution of the world on 3 threads

A simple parallel schedule for the computation of a single iteration of the game of life can be devised by splitting the task in two: initially all threads collect the two border lines they need for the computation from their neighbors, and subsequently they compute the next state for their part of the grid. For the first part of the schedule's flow graph, the two border lines can be collected by using a simple split-process-merge construct illustrated in Figure 18. The *split border requests* split operation creates two *border request* output data objects that are routed with an appropriate routing function respectively to the thread holding the cells above the local block, and to the thread holding the cells below the local block. The *copy border data* leaf operation copies the requested border data into an output data object. Finally the *merge border data* merge operation copies the border data into the additional memory available for the borders in the local thread storage.

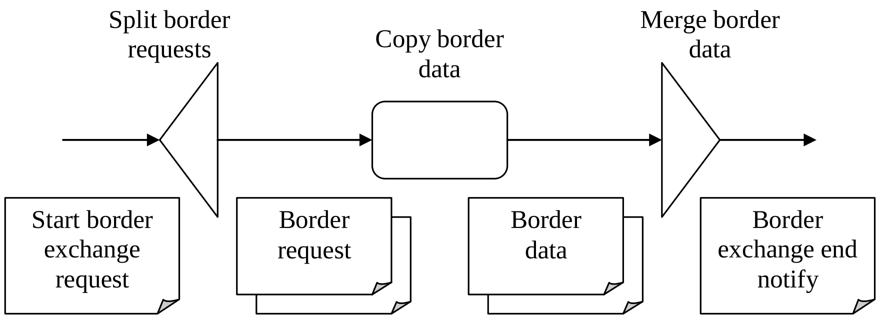


Figure 18. Flow graph segment for border exchange

Once this part of the schedule is completed, the next part must compute the new state of the local cells based on the current state and the borders that have been collected. The flow graph for this part of the application is composed of a single leaf operation that performs the computation. The new state of the cells replaces the previous state in the thread local storage.

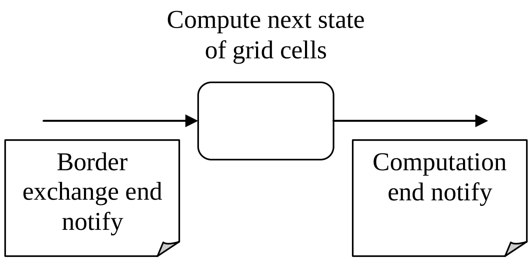


Figure 19. Flow graph segment for computation of next state of local cells

In order to complete the parallel application, it is necessary to execute this sequence of operations in parallel on all the threads on which the grid of cells is distributed. This can be achieved by enclosing the previously described sequences within a split-merge pair that sends out requests to all the threads that store part of the grid and collects the notifications when the operations have completed execution. The intermediate global synchronization caused by the merge operation placed after the border exchange operations ensures that all nodes have completed the border exchange before computation starts. This additional synchronization is required since the new state of the cells will replace the previous state required for the border exchange.

The final merge serves as a barrier to synchronize the end of the iteration on all nodes, since it will not return its output data object before all the enclosed branches have completed their computations. The complete flow graph is illustrated in Figure 20.

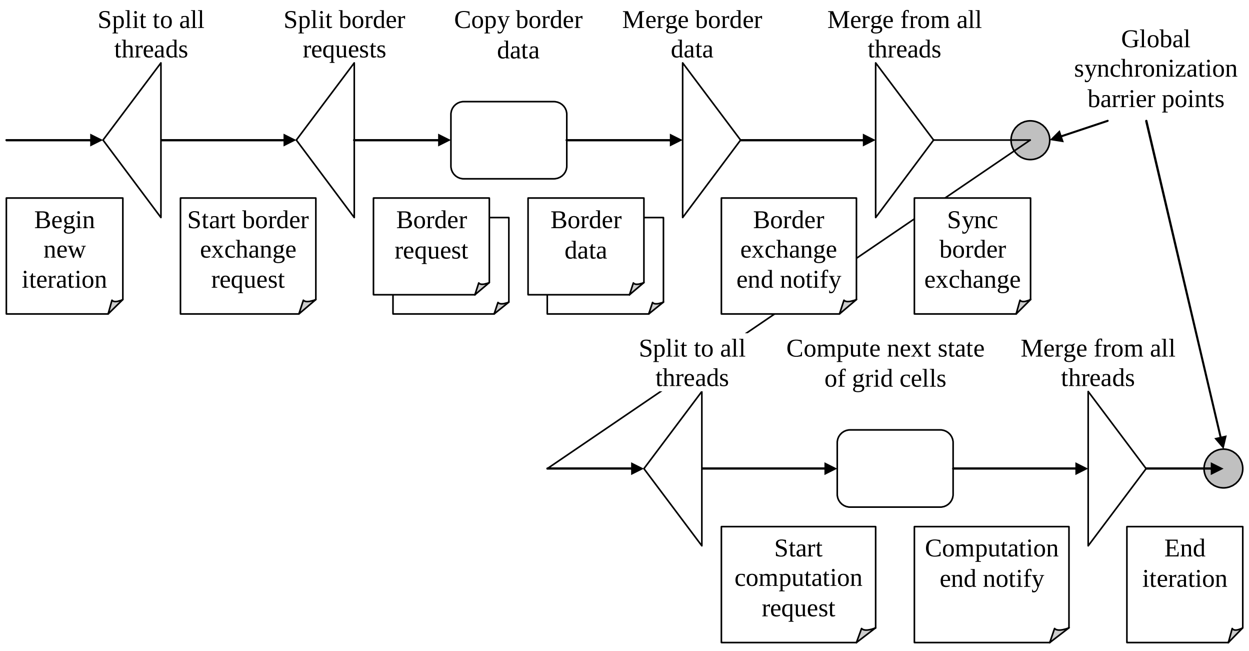


Figure 20. Complete flow graph for the game of life

In order to complete the parallel schedule, we need to assign thread collections to the operations contained in the flow graph. Like the compute farm example, two thread collections are used. The first is used for the execution of the global split-merge pair at the extremities of the flow graph, and contains a single thread. The other is used to store the distributed state of the grid and for performing all the other operations (border exchange and new state computation). The complete parallel schedule can be represented using an unfolded variant of the flow graph, showing the thread collections and the individual threads on which the operations are executed. On this unfolded view, every flow graph edge represents one data object that triggers the execution of the operation at its extremity.

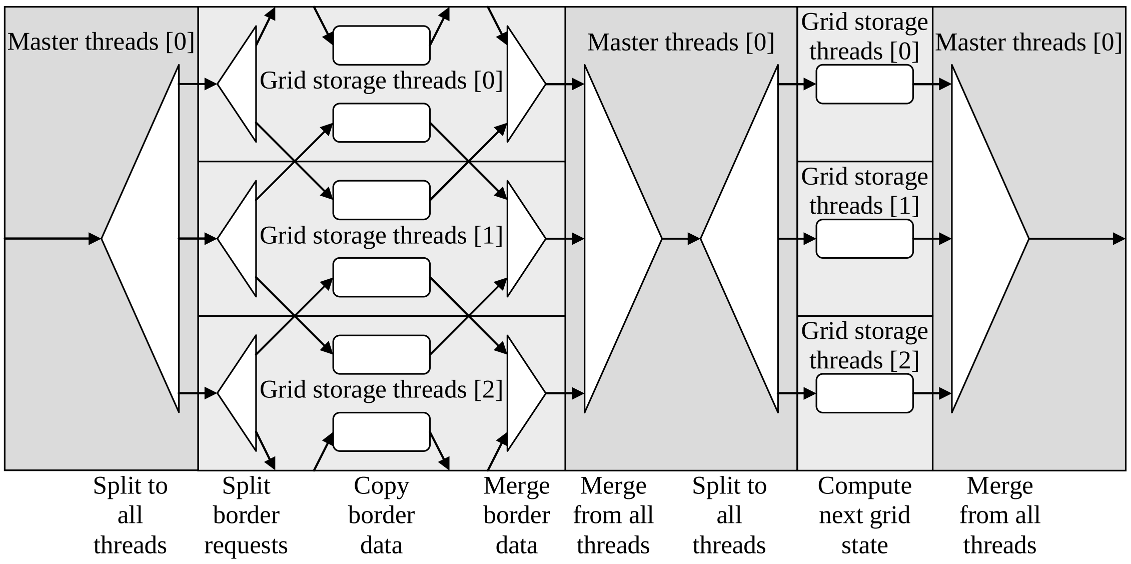


Figure 21. Unfolded flow graph view for the parallel schedule of the game of life, showing the thread collections and individual threads.

Since the application pattern of the game of life (distributed data storage, iterative execution with border exchange) is very common, this parallel schedule and variants thereof are presented in greater detail in Chapter 4.

2.5 Runtime behavior of parallel schedules

The previous sections have shown how to build a parallel schedule for a parallel application. This section focuses on the runtime behavior of parallel schedules. Let us assume a simple execution model where each parallel schedule thread is assigned to a distinct operating system thread located on a separate processor. All parallel schedule threads thus run totally independently of one another, without any resource contention. We also assume that communication between threads takes time, i.e. sending a data object from one thread to another is not instantaneous, but sending a data object from the current thread to itself is free. This assumption is made on the basis that parallel schedules will usually be executed on clusters of computers, where network communications are required to move the data objects from one node to another. Finally, we assume that the underlying hardware allows for simultaneous computation and communication.

2.5.1 Data object queues

In order to execute parallel schedules, we use data object queues located in the individual threads. For instance, let us consider the execution of the simple compute farm parallel schedule presented in section 2.3, in a situation where the split operation creates 6 subtask data objects. Figure 22 illustrates the fully unfolded parallel schedule view, showing all the operations that are executed during program execution and all the data objects. Every oriented edge in the unfolded view of the parallel schedule represents a data object, ending in the operation that processes the data object. Since the fully unfolded view can easily become overcrowded if the application creates large numbers of data objects, and therefore executes many operations, we often use a partially unfolded view showing only a single instance of each operation for every thread. On the partially unfolded view in Figure 22, the data objects are also shown, illustrating that each operation is executed twice on each thread.

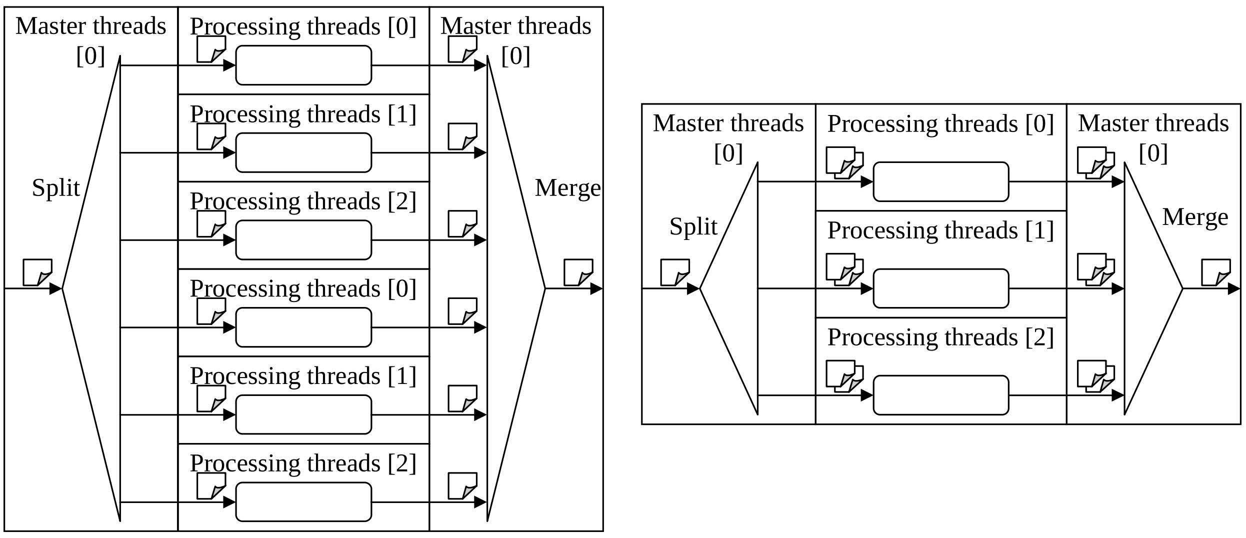


Figure 22. (left) Fully unfolded view of a simple compute farm parallel schedule and (right) partially unfolded view of a simple compute farm parallel schedule showing data objects

Since a thread can only execute one operation at a time, the partially unfolded view is actually a representation of the runtime behavior of the parallel schedule. For instance, when the execution of the parallel schedule is started by the initial request data object, the split operation starts execution, and produces 6 output data objects. These output data objects are distributed onto the 3 processing threads by using the round-robin routing function, each thread receiving two data objects. The incoming data objects are stored within the thread's data object queue. As soon as a data object is available within a thread's data object queue, that thread immediately starts processing by executing the appropriate operation. When the execution of the operation completes, the next data object within the queue is retrieved for processing. The arrival of the output data objects from the split operation will thus cause the three processing threads to start execution in parallel, and to send their output result back to the master thread, which initiates the merge operation. The timing diagram for the complete program flow is shown in Figure 23.

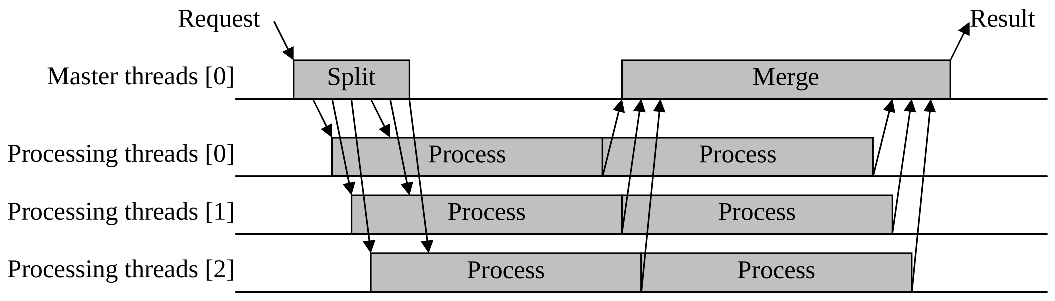


Figure 23. Timing diagram for a simple compute farm parallel schedule, where the arrows represent circulating data objects

Since data objects are sent to the target thread as soon as they are created by an operation, their transfer can in many cases be overlapped with ongoing computations. This behavior also ensures that the data object queues of threads usually contain some data objects, allowing computation to continue uninterrupted. The latencies and transfer times related to network communications are thus hidden.

2.5.2 Flow control

Since the parallel schedule execution model sends data objects as soon as they are created, a split operation that creates many output data objects might cause large amounts of memory to be allocated within the target threads in order to store these data objects, as well as induce a high network load in order to send all these data objects. In order to resolve this problem, parallel schedules provide a flow control mechanism that limits the number of data objects that can be in circulation at a given moment. The limitation is achieved by introducing a feedback loop between a split-merge operation pair. The split operation is initially permitted to emit a fixed number of data objects, after which it is suspended. For each data object that is received by the corresponding merge operation, the split operation may emit a new data object.

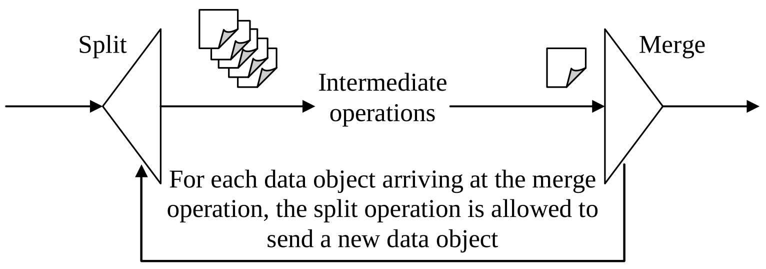

Split
Intermediate operations
Merge
For each data object arriving at the merge operation, the split operation is allowed to send a new data object

Figure 24. Flow control in parallel schedules

In order to limit the additional communications induced by the feedback loop, it might not be desirable to send individual notifications to the split operations in cases where the split operations create very high numbers of small data objects. It is possible to specify a number of data objects that must be received by the merge operation before a notification is sent back to the split operation.

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 data object queues during execution. When the data queues are empty, the processors are idle while waiting for a new data object to arrive, leading to under-utilization of the available resources. Care should be taken to set the value so that every thread will always have at least two data objects: one being processed within an operation, and another waiting for processing.

2.5.3 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 to each of them. The distribution of tasks is controlled by the routing function attached to the first operation after the split operation in the flow graph. In order to achieve load balancing, this routing function has to select threads based on their effective computation throughput, thus sending more data objects to the threads that can process more operations.

Such a routing function can be constructed automatically by taking advantage of the feedback loop introduced for the flow control mechanism. Instead of maintaining a constant number of data objects in circulation between the split operation and merge operation on a global scale, the number of circulating data objects is maintained constant for each thread that can be reached from the split operation. This is achieved by storing the index of the path used for routing within the data objects sent by the split operation, and preserving this value until the data object arrives at the merge operation. The merge operation then returns this index to the split operation in its flow control notification. The next data object created by the split operation will be sent along this path, thus maintaining the number of data objects on each path constant. In order to initialize the system, the routing function initially routes the data objects according to a round-robin scheme up to the limit imposed by flow control.

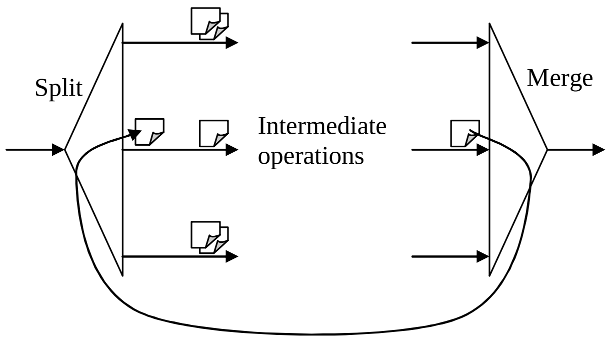


Figure 25. Load balancing in parallel schedules. When the split operation sends out a new data object, it is sent to the thread that last returned a data object to the corresponding merge operation.

The load balancing mechanism can effectively equilibrate uneven computation loads provided that the number of data objects representing subtasks is significantly greater than the number of threads used for computation.

2.6 Conclusion

This chapter presented the concept of *parallel schedules* as a means to describe parallel applications. A parallel schedule is composed of two parts: a flow graph, and a set of thread collections. The flow graph provides a complete model of the program flow of a parallel application by interconnecting individual sequential operations of four basic types: leaf, split, merge, and stream. Flow graphs can contain any combination of operations, as long as the fundamental symmetry of split and merge operations is preserved. By using a flexible looping construct in order to repeat sections of the flow graph, complex pipelined or nested sequences of operations can be created within flow graphs.

The thread collections provide sets of *threads*, which provide the execution context within which the operations are executed. These threads are mapped at runtime onto processing nodes, enabling parallel execution of the application.

The execution model of parallel schedules is data driven and fully asynchronous, enabling maximum utilization of the underlying hardware resources. Additional mechanisms such as flow control and load balancing are provided by the parallel schedules model in order to give the developer more control over the flow of data objects within the parallel schedule. The flow graphs and the corresponding thread collections can be represented graphically at various levels of detail, providing an intuitive approach to the understanding and design of parallel applications.

Chapter 3

Dynamic Parallel Schedules

Dynamic Parallel Schedules (DPS) is a new parallel programming library based on the parallel schedules approach. DPS includes features such as advanced pipelining constructs, the ability to share flow graphs among multiple applications running on the same cluster, and the possibility for an application to change its mapping to processing nodes during execution. This chapter presents an overview of the DPS library and its application programming interfaces (APIs).

3.1 Introduction

A first generation parallel schedules implementation, called Computer-Aided Parallelization (CAP), was developed in the mid-90s [Gennart99][Messerli99]. The present implementation of parallel schedules, DPS, builds on this previous experience, and extends the model with new concepts and functionality. The presentation of the parallel schedules model in the previous chapter is based on the current implementation in DPS.

This chapter focuses on the implementation of parallel schedules within DPS, and the new functionalities that are provided. First and foremost is the dynamic aspect. We want to allow almost all parallel program structures to be modifiable at runtime. This includes the possibility to allocate and release processing nodes during execution, allowing the application to adapt the number of processors it uses to its current execution requirements. It also includes the ability to include or exclude parts of the flow graph based on runtime decisions. The dynamic usage of processing nodes also serves as a basis for implementing transparent fault tolerance in applications, detailed in Chapter 6.

Another important aspect was to implement parallel schedules using a simple C++ library, since this dramatically lowers the learning curve and allows developers to keep their favorite development environments. Not relying on any type of custom preprocessing obviously restricts the possibilities to the limitations of the C++ language [Ellis90], where the lack of features such as data type reflection makes fully transparent serialization impossible. Most of the language related aspects of the DPS library are handled in Chapter 5, which discusses the internals of the DPS library.

Finally, we wanted to improve the parallel schedule model by making it more coherent, providing a more uniform programming model. New structures were also introduced to allow for improved pipelining in applications, since this has a major effect on performance. We also have included the ability for multiple applications running on the same cluster to cooperate by sharing parts of their flow graphs with one another. This ability allows an application to serve as a *parallel component* for another application by providing parallel services.

This chapter presents the features that make DPS a novel and intriguing parallel application development framework from a software developer's perspective. The concepts used in DPS applications will be reviewed, and illustrated in the context of real applications. The initial sections of this chapter present the general concepts of the implementation of parallel schedules in DPS. The specificities of the implementation of parallel schedules as a C++ library are detailed starting from section 3.5. Finally, a short summary of the major differences with the previous CAP implementation is provided in section 3.7. A brief description of the CAP syntax as well as a comparison with the DPS syntax can be found in Appendix A.

3.1.1 Dynamic node allocation

Whoever has used MPI will likely be familiar with the following lines of C code:

```
int rank, size;
MPI_Comm_rank( MPI_COMM_WORLD, &rank );  // Local node ID
MPI_Comm_size( MPI_COMM_WORLD, &size );  // Number of processing nodes
```

These are the instructions where the application discovers how many processing nodes it has been assigned, and where it is located within these nodes. This information stays constant for the whole duration of program execution. If the application requires more or less processing power for a specific task during its execution, there is nothing it can do to change these assignments. However, many parallel applications do not have constant requirements during execution, whether it is because they contain lengthy sequential parts interspersed between the parallel segments, or because they cannot efficiently use a constant number of processors over their complete execution cycle.

The requirement for dynamicity may not necessarily lie within the application itself, but also within the cluster used for running the application. A popular means to achieve cheap high performance computing is to exploit the unused processing power of large numbers of desktop PCs. This technique has been successfully used by several large projects, such as SETI@home [Anderson02] or Folding@home [Pande03]. Since the computers used in such computations are not entirely dedicated to the parallel processing task, the parallel application needs to support dynamic disconnection and connection of processing nodes. This type of cluster needs not be limited to the Internet at large. The desktop PCs may also be those of a company or a university. Most of the performance measurements presented in this thesis were actually performed on a resource of this type: a room full of identical workstations used for teaching during the day, but often available at night.

3.1.2 Dynamic application flow

Besides the dynamicity needed for allocating or releasing computing resources, dynamic program execution can also be required based on the input data which the application needs to process. One possibility that can be used in order to achieve dynamic execution would be to use a static flow graph containing all possible operations, and at runtime skip the unneeded operations with *if* clauses placed within the sequential operation declarations. However, the source code is usually more readable and maintainable if these clauses appear at a higher level within the parallel program description, by dynamically constructing a flow graph at runtime specifically tailored for the task at hand.

3.1.3 Heterogeneous execution environments

Another important aspect of the DPS library is the support for program execution on heterogeneous clusters. This requirement implies that the DPS library itself is cross-platform, allowing it to be compiled on various processor architectures and operating systems. The network protocols used by the DPS library for transferring data objects also need to be transparent to platform differences such as data endianness and structure alignment.

3.2 Flow graph construction

DPS uses flow graphs to describe the execution flow of parallel applications. Unlike many other frameworks and languages, the flow graph does not need to be statically declared within an application's source code. The source code provides a collection of statically defined components such as operations or routing functions that are assembled at runtime to build the final flow graph. The flow graph can subsequently be used to execute parallel schedules. Flow graphs can be assembled at any time during execution, and can be used to spawn any number of parallel schedules.

Since DPS is a C++ library, all of the elements that are used to build flow graphs are implemented within the application's source code as C++ classes. The syntax of these elements is presented starting from section 3.5. The following sections present the elements that used to build flow graphs in detail.

3.2.1 Connecting operations

Operations are the nodes of the flow graph, encapsulating all of the application's functionality. Within DPS, all operations share a common syntax and programming model. DPS provides four basic operation types: *leaf, split, merge* and *stream* operations. The stream operation is a new concept introduced in DPS, and can be used to improve the pipelined behavior of applications.

The body of an operation is composed of standard sequential C++ code that performs the operation's tasks. Operations are further characterized by three external parameters: the acceptable input data object types, the produced output data object types, and the type of thread used for local data storage. When a flow graph is built, the sequence of operations is validated at compile time to ensure that all connected operation pairs have at least one common data type. The validation checks for a simple flow graph are illustrated in Figure 26.

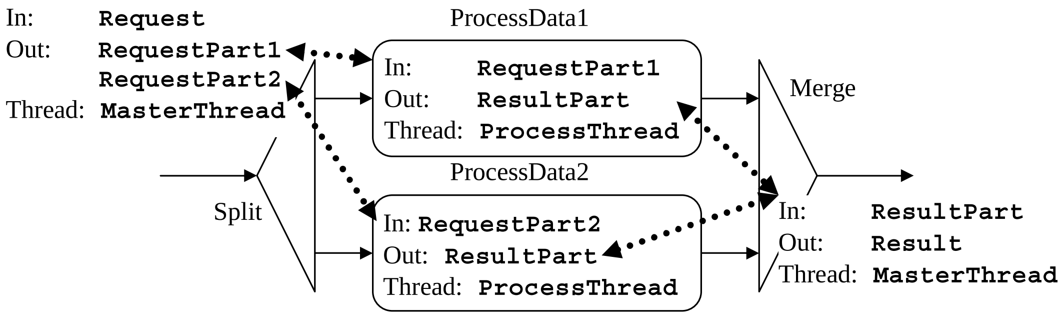


Figure 26. Assembling a flow graph. The dotted arrows represent the type validation checks that are performed for the ProcessData operations in order to ensure that the flow graph is valid.

Flow graphs can also contain loops, used for replicating sequences of operations at runtime. When inserting loop constructs, two constraints need to be respected. The first constraint is that the input data object of the first operation within the loop construct is the same as the output data object of the last operation, in order to ensure that the output data object can be looped back as input. The other constraint is to ensure that the flow graph keeps the symmetry between split and merge operations. The first constraint can be validated at compile time, however the second constraint can only be checked at runtime, since the condition that determines the number of repetitions is evaluated at runtime.

3.2.2 Attaching thread collections

In order to provide the execution context for the operations, these operations need to be attached to thread collections. The thread collections contain a set of threads of a user-defined type, which can be used for example to store parts of a distributed data structure. As part of this attachment, a routing function is also assigned to the operation. The routing function performs the selection of the thread within the thread collection on which the operation should be executed. As with the interconnection of operations, 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 data object type in order to select the target threads. Many routing functions are not data dependent, and therefore do not care about the data object type that is to be routed. Similarly, 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. For these cases, any data object type or thread type will pass the validation.

The additional validations for thread collections and routing functions are illustrated in Figure 27. The illustrated flow graph splits a task on a distributed data structure stored in a collection of threads of type *DistDataThread*. The routing function for the processing operation is a user defined data dependent routing function, *DataDependentRoute*. Therefore, for this operation, both the thread type and routing function require validation.

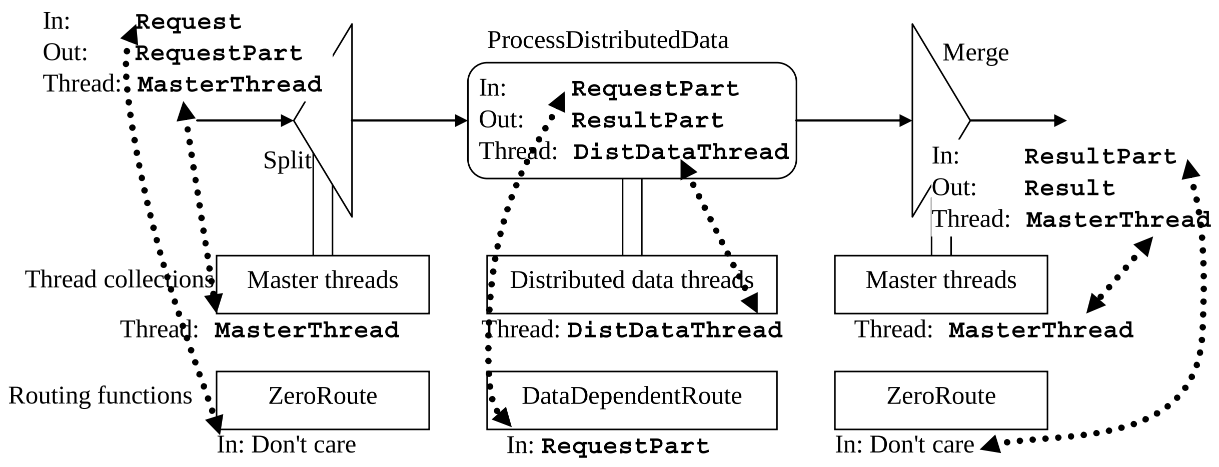
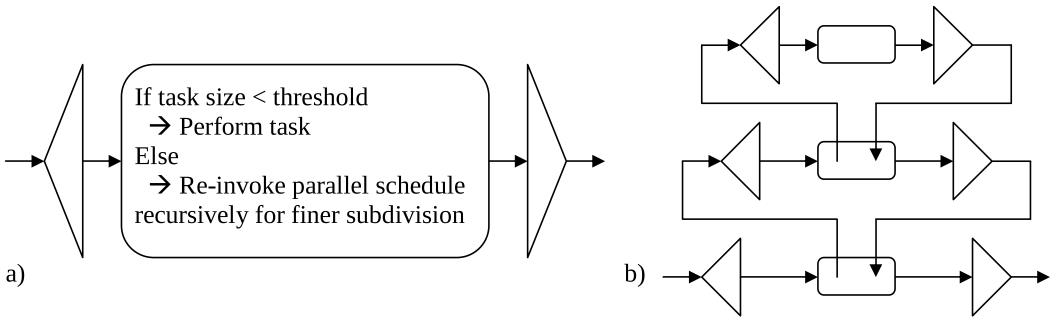


Figure 27. Assembling a flow graph. The dotted arrows represent the type validation checks that are performed for the operations in order to ensure that the flow graph is valid.

3.2.3 Parallel schedule invocation

A parallel schedule does not need to be fully self-contained. DPS supports the ability to invoke parallel schedules from within the execution of another parallel schedule, in particular also recursive invocations. A simple parallel application could for example recursively invoke itself until the data remaining to be processed is of a reasonable complexity to be processed on a single node, as illustrated in Figure 28.



If task size < threshold
→ Perform task
Else
→ Re-invoke parallel schedule recursively for finer subdivision
a)
b)

Figure 28. a) Recursive invocation of a simple parallel application and b) flow graph of the resulting execution

These recursive parallel schedule invocations are ideal for some specific problem types, such as tree traversals which are commonly found in many algorithms. The split operation generates one data object for every branch on the tree leaving the current node. The leaf operation then either re-invokes the parallel schedule if the node at the end of the branch has further children, or performs the required processing. The merge operations subsequently combine the results moving back up the tree to the root node. Invoking parallel schedules is an inexpensive operation as long as the same flow graph is reused, allowing the recursive approach to be efficient. However, managing the routing and load balancing in recursive parallel schedules is a non-trivial task.

3.3 Dynamic thread collections

The parallel schedule approach relies on threads, thread collections, and routing functions to describe the mapping onto processing nodes of the various operations that compose an application. When a thread collection is initially created, it does not contain any threads. The contents of the thread collection are handled dynamically and can be altered at any time, even when parallel schedules are running on the threads within the collection. A thread collection supports three basic operations on its threads: adding new threads, removing existing threads, and moving existing threads from one processing node to another.

Dynamic thread collections are a very powerful mechanism for handling tasks such as load balancing or load distribution. An application can add or remove nodes from its working set without stopping and restarting computations. For applications that do not store any data within their threads, standard routing functions such as round-robin or the built-in load balancer will automatically use all the available threads. When the threads are used to store parts of a distributed data structure, the application developer needs to make sure that the data structure is kept coherent when adding or removing threads.

When threads are moved from one node to another, the operation is fully transparent, since the location of the threads is never exposed to the application. The DPS framework takes care of redirecting all communications to the new thread location, and of ensuring that the current thread state remains consistent throughout the transfer.

A typical usage pattern for moving threads from one node to another within a DPS application is shown in Figure 29. The application uses thread movement to perform dynamic load balancing on a static data distribution. The data distribution is initially divided into 16 parts, evenly distributed onto 4 nodes. Each part of the data structure is stored within a DPS thread, with 4 DPS threads on each node. If, after running for some time in such a configuration, a load imbalance appears, the application can simply move some of the threads from the overloaded nodes to the less loaded nodes.

Another usage is to dynamically adjust the number of nodes an application is currently using, for example based on its computation requirements or on the availability of nodes. Additional nodes can be exploited by moving some existing threads onto these nodes, for example when a data structure is divided onto a fixed number of threads. In other circumstances, it might also be possible to simply create new threads for the additional nodes, and redistribute the data structure onto all the available threads.

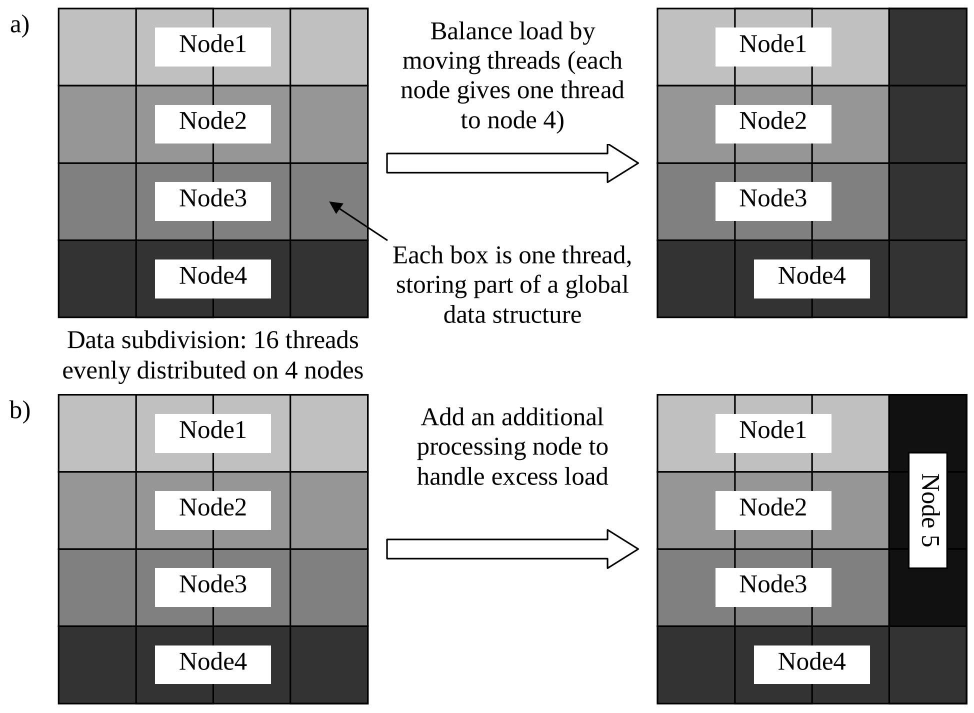

a)
Node1
Node2
Node3
Node4
Balance load by moving threads (each node gives one thread to node 4)
Each box is one thread, storing part of a global data structure
Data subdivision: 16 threads evenly distributed on 4 nodes
b)
Add an additional processing node to handle excess load
Node 5

Figure 29. Dynamic thread collections: a) Load balancing by moving threads from one node to another, b) Adding an additional node to handle some of the threads

3.3.1 Defining thread collections

DPS thread collections have several attributes that are shared by all the threads contained in the collection. The most important attribute is the *thread type*, which defines the data type that is stored within threads of this collection. The other attributes affect the mapping of the threads of this collection to operating system threads for executing operations.

The first of these attributes indicates how many operating system threads should be created for every DPS thread, thereby limiting the number of operations that can be simultaneously executing within the context of that DPS thread. The other attribute indicates whether the DPS threads should be aggregated onto the same set of operating system threads when multiple DPS threads are mapped onto the same node. The aggregation mechanism ensures efficient execution in the common case where DPS applications map several threads of the same type onto the same processing node (for example the distributions shown in Figure 29). Running multiple DPS threads on one single-processor node by using multiple operating system threads for executing their operations would be inefficient, since the operating system would needlessly waste time to perform context switches between the various threads.

Aggregation can be used in conjunction with the multiple operating system thread option; in this case there will be exactly the number of requested operating system threads on each node. The various combinations of aggregation and multiple operating system threads are illustrated in Figure 30. An example for the use of multiple operating system threads per DPS thread is given in Chapter 4.

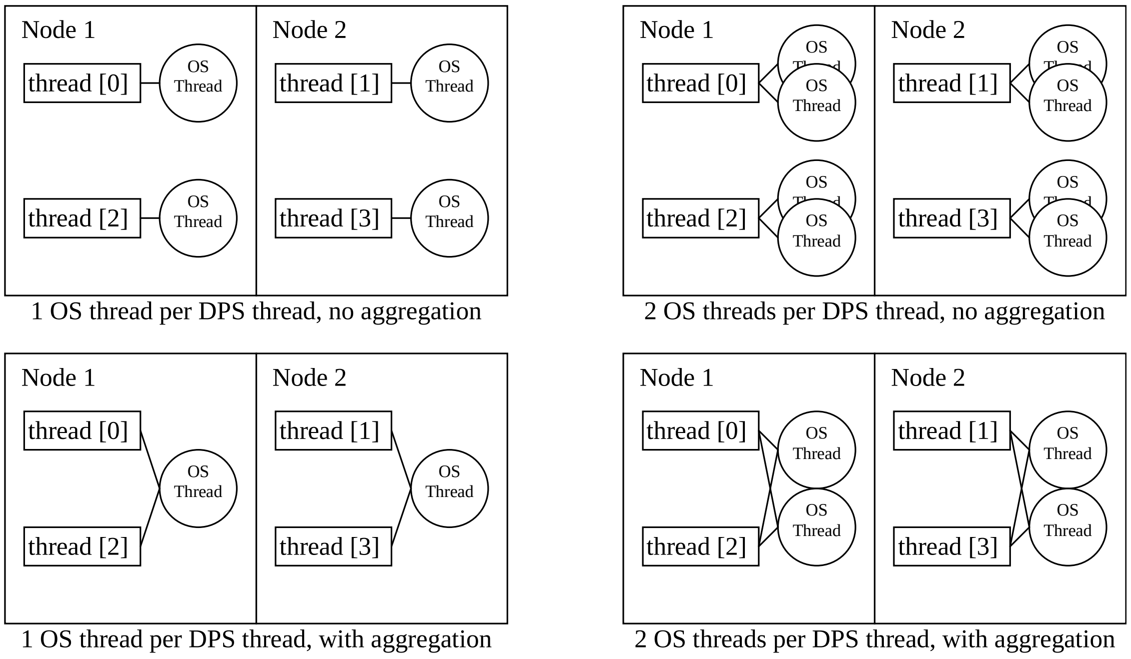

Node 1
Node 2
thread [0]
thread [1]
thread [2]
thread [3]
OS Thread
1 OS thread per DPS thread, no aggregation
2 OS threads per DPS thread, no aggregation
1 OS thread per DPS thread, with aggregation
2 OS threads per DPS thread, with aggregation

Figure 30. A thread collection with 4 threads spread on 2 nodes, using all combinations of aggregation and multiple operating system threads.

3.4 Parallel components

Interaction between multiple running parallel applications is an area of growing interest in parallel computing. Inter-application interactions are common in distributed business applications using frameworks such as CORBA [OMG02] or DCOM [Microsoft96], or the newer features such as web services provided by the Java [Gosling96] and .NET [Platt03] platforms. These interactions focus mainly on point to point communications between two applications. Parallel applications, however, usually deal with distributed data structures spanning multiple nodes, requiring 'many to many' communication patterns rather than the classical 'one to one' approach.

3.4.1 Related work

Parallel components can satisfy various needs. A common case is the interconnection of several simulations of specific systems in order to enable the simulation of a larger, global system. Such a system is typically composed of a series of separate applications, where the output of one application serves as input for the next application. Each participating applications uses its own internal data structures, optimized for its particular computation tasks. The participating applications may also be running on distinct sets of computation nodes. The most common challenge is therefore to rearrange the distributed data structure used by one application running on *M* nodes into the distributed data structure used by another application running on *N* nodes, known as *MxN* [Bertrand05]. An example of such a redistribution task is illustrated in Figure 31. For this type of multi-application integration, the Common Component Architecture forum was created, in order to specify an interaction framework [Armstrong99].

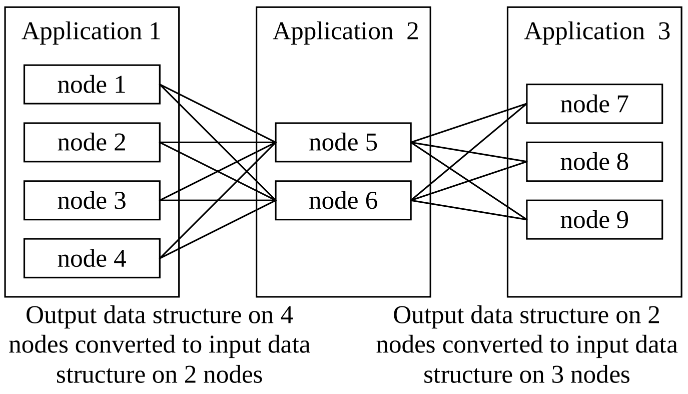

Application 1
Application 2
Application 3
node 1
node 2
node 3
node 4
node 5
node 6
node 7
node 8
node 9
Output data structure on 4 nodes converted to input data structure on 2 nodes
Output data structure on 2 nodes converted to input data structure on 3 nodes

Figure 31. Redistribution of data from one parallel application to another parallel application (MxN)

The redistribution of data from one layout to another might also imply some complex transformations, since many applications do not simply store their state in simple flat structures such as arrays or matrices, but use complex structures such as octrees. These data structures also need to be constructed properly during the redistribution process.

Many approaches have been proposed for meeting this challenge, including Meta-Chaos [Edjlali97] and InterComm [Lee05]. These frameworks use data descriptors in order to enable parallel applications to specify their data layout, and can generate optimized data transfers from one application to another.

3.4.2 Parallel components in DPS

DPS takes a different approach to parallel components for applications written within the DPS framework. Rather than externally connecting separate DPS applications, the library provides the ability to dynamically combine flow graph parts from the various participating applications into one single flow graph. The resulting flow graph can be used to execute a single parallel schedule spanning all applications.

These flow graphs can be of two types: the forward-flowing form illustrated in Figure 31, where the program flow moves from one application to the next as the execution progresses, or the bidirectional form where data is moving back and forth between the applications. This second form is typical for services, where one application makes a request to another application, and later receives the results. An example of such services is the access to a striped file spread across several nodes. Figure 32 illustrates a simple case where two applications access a striped file service. Another example is a service providing insight into the current state of an ongoing computation, which will be presented in Chapter 4.

Using parallel components rather than libraries for services such as file I/O is useful since the parallel component can share critical resources such as an in-memory file cache amongst multiple applications or for subsequent executions of the same application.

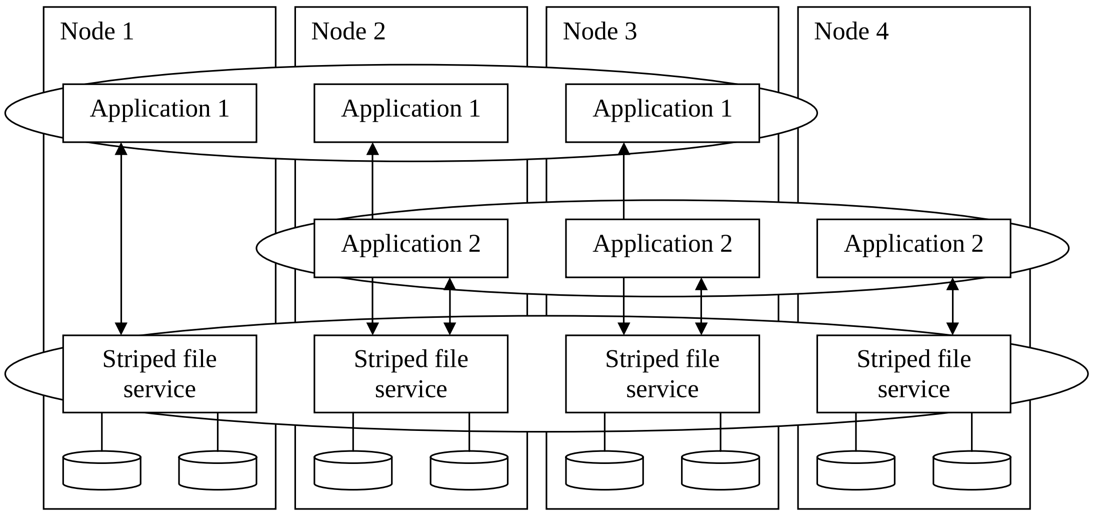


Figure 32. Multiple applications accessing a shared parallel component

3.4.3 Designing parallel component schedules

For example, the striped file service shown in Figure 32 provides a flow graph section to applications that want to read data from a striped file. The flow graph section, shown in Figure 33, takes as input data object a request indicating which parts of the file should be read. The split operation analyzes this request and sends out individual requests to the nodes that have the data in their local storage. The data is read from the disks, and sent back in parallel to the calling application. Flow graphs that are exposed as parallel components need not satisfy the usual split-merge symmetry requirements of parallel schedules, since the missing parts are provided by the application that uses the parallel component.

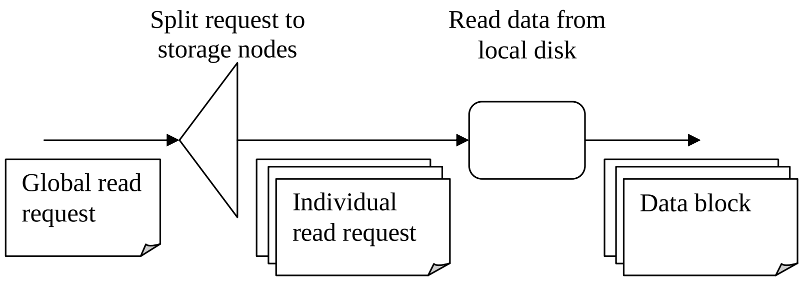


Figure 33. Flow graph section for a striped file parallel read operation

The flow graph section shown in Figure 33 uses two thread collections. The split operation is running on a thread collection with a single thread, with the thread local data containing sufficient information for the split operation to distribute the request onto the storage nodes. The read operation is running on a thread collection with one thread per disk in the cluster. Each of these threads stores a cache of previously read data blocks in order to accelerate future accesses. Since DPS ensures that each thread is only executing one operation at a time, there is no problem with reentrance in the read operations even when the service is used by multiple parallel applications simultaneously.

An application using the service can integrate this flow graph section into its own flow graph, as shown in Figure 34. The operations belonging to the application are executed in the application's threads, whereas the operations belonging to the parallel service are executed in the parallel service's threads. The data objects are transparently transferred from one application to another. The use of partial graphs in parallel components is important for optimal performance, since it allows the large data objects that result from the disk read operations to stay within the same node for processing, as illustrated in the unfolded parallel schedule shown in Figure 35.

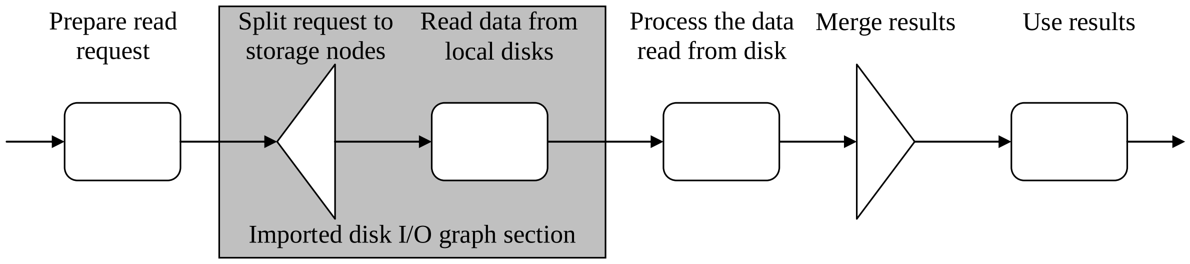

Prepare read request
Split request to storage nodes
Read data from local disks
Process the data read from disk
Merge results
Use results
Imported disk I/O graph section

Figure 34. Flow graph using a parallel service to read from a striped file

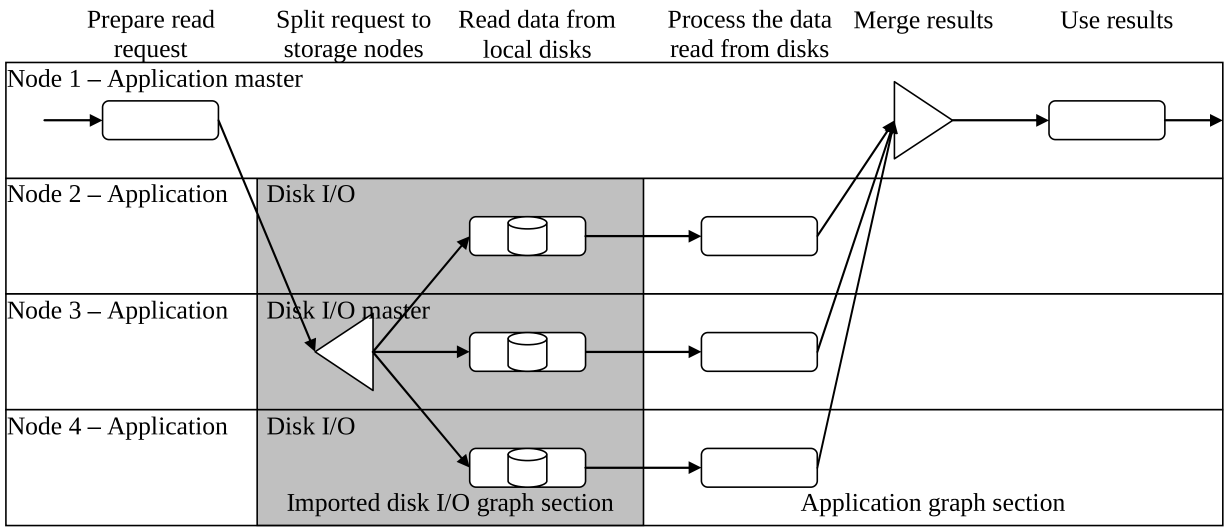

Prepare read request
Split request to storage nodes
Read data from local disks
Process the data read from disks
Merge results
Use results
Node 1 – Application master
Node 2 – Application
Disk I/O
Node 3 – Application
Disk I/O master
Node 4 – Application
Disk I/O
Imported disk I/O graph section
Application graph section

Figure 35. Parallel schedule using a parallel service (Disk I/O graph section) in order to read data from a striped file

3.4.4 Example usage

A parallel striped file system service is useful for example in visualization applications involving very large datasets, in particular when the data needs to be processed before being displayed and multiple users access the system simultaneously. An example of such a system is curved surface extraction from anatomical datasets [Saroul03]. While current anatomical datasets such as the Visible Human [Ackerman98] easily fit onto a single personal computer with their sizes between 20GB and 50GB, future datasets acquired using high resolution imaging techniques can easily consume terabytes of space. For high-throughput data visualization, data needs to be read out in parallel and processed in preference on the node on which it resides in order to minimize network usage.

Such large volume data sets can be split into small cubes forming a multi-resolution representation of the complete dataset [Gerlach02]. These small cubes are subsequently distributed onto the storage nodes, and managed by the striped file system component. Interfacing parallel applications to the parallel striped file system component has the advantage of considerably speeding up the access to the lower resolution cubes, since they will most likely stay in the cache from one access to the next. The complete flow graph for the parallel extraction of curved surface parts is illustrated in Figure 36.

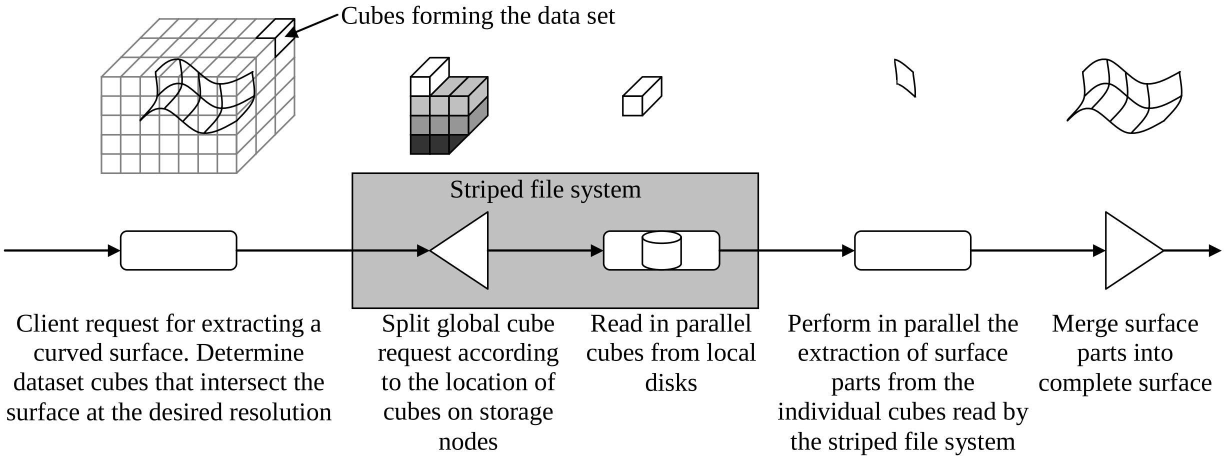

Cubes forming the data set
Striped file system
Client request for extracting a curved surface. Determine dataset cubes that intersect the surface at the desired resolution
Split global cube request according to the location of cubes on storage nodes
Read in parallel cubes from local disks
Perform in parallel the extraction of surface parts from the individual cubes read by the striped file system
Merge surface parts into complete surface

Figure 36. Curved surface extraction from a volume dataset using a striped file system parallel component

3.5 Implementing DPS applications

The previous sections described the features supported by Dynamic Parallel Schedules (DPS). The following sections focus on the application development model provided by the DPS class library and show how a developer can use DPS features.

Since the DPS framework is a C++ library, it follows that language's common usage patterns. All the declarations for the framework are included within the *dps.h* header file, and contained within the *dps* namespace. The namespace specification is present in all the following sample source code, but omitted within the text in order to improve readability. Since the C++ language provides no real support for features such as data reflection, DPS incorporates its own mechanisms to provide this functionality. The missing functionality is mostly provided by macros that are placed in the source code. The most prominent of these macros is the *IDENTIFY* macro, which is used for generating cross-platform runtime type information and class factory management. The macros and their inner workings are detailed in Chapter 5.

3.5.1 The DPS Controller

DPS uses a *Controller* object to keep track of everything present in the execution environment. The Controller is responsible for managing all the resources used by a parallel schedule: the thread collections and the threads they contain, the flow graphs, and the state of the currently executing parallel schedules.

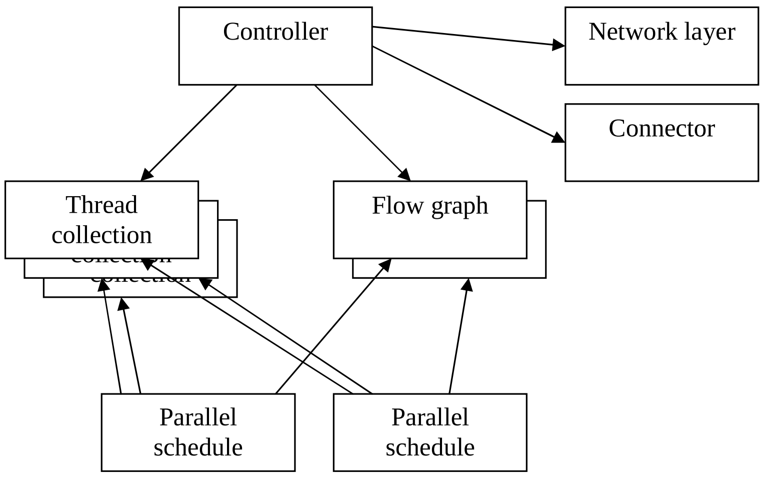


Figure 37. Overview of the DPS object model

The Controller also owns some helper objects, such as the network layer used for communicating with other nodes, and the connector used to execute remote instances of the parallel application.

The controller object also exposes the interfaces to the DPS library for tasks such as creating thread collections, creating flow graphs, or executing parallel schedules. Most DPS objects provide a method to obtain a pointer to the application's controller:

```
dps::Controller *controller = getController();
```

3.5.2 Applications

DPS applications derive from the *Application* base class, which contains the startup functions of the application. Since DPS applications are distributed on many nodes, DPS provides two startup functions: *init* and *start*. The *init* function is called on all nodes, whereas *start* is only called on the node where the application is invoked. Both functions are called after the initialization of the DPS controller, and can therefore access all the features of the DPS library. Like all classes used within the DPS framework, the IDENTIFY macro is used to provide basic type reflection features.

```
// Simple application class sample
class SimpleParallelApp : public dps::Application
{
public:
  // Application startup function
  virtual void start();

  // Application initializazion function
  /* Returns true if successful, false if initialization has
     failed. DPS will quit on failure.
  */
  virtual bool init();

  IDENTIFY(SimpleParallelApp);
};
```

The startup function *start* typically contains the code required to create the thread collections and the flow graph, and is also responsible for running the parallel schedule. The initialization function *init* is typically used to parse command line parameters and read configuration files. The application class is also an ideal place to store globally accessible variables, since a pointer to the application object can easily be obtained from within the context of most other DPS objects through a *getApplication* member function.

3.5.3 Thread collections

The threading model of the parallel schedule approach shown in section 2.3 is implemented in DPS by using a simple two level hierarchy: *thread collection* objects containing an arbitrary number of *thread* objects. These objects are created using a simple class model. Threads are user-defined C++ classes representing the thread local storage. A thread collection is a variable-sized distributed container of threads. If the operations running within a thread collection do not require access to any locally stored data, DPS provides a specific thread class *StatelessThread* that can be used to indicate that the threads do not store any local state.

Since thread collections are simply containers for threads, creating a simple stateless thread collection requires only a single parameter: the name of the thread collection.

```
dps::StatelessThreadCollection processThreads =
  getController()->createStatelessThreadCollection("process");
```

In order to create a more complex thread collection controlling the aggregation, the operating system thread mapping and the thread type, several additional arguments can be specified. The thread type is provided as a template argument in order to allow validation of the type at compilation time.

```
dps::ThreadCollection<StorageThreadType> processThreads =
  getController()->createThreadCollection< StorageThreadType > (
    "process", // name
    2,         // Number of OS threads per DPS thread (default 1)
    false      // Enable aggregation (default true)
  );
```

The *ThreadCollection* object provides methods for managing the threads it contains. Threads can be added, removed and moved from one node to another. Before the thread collection can be used for executing parallel schedules, it needs to contain at least one thread. The following methods are used for updating the thread collection:

```
// Add a new thread
processThreads.addThread("hostname");

// Remove a thread
processThreads.removeThread(threadId);

// Move a thread
processThreads.moveThread(threadId,"newhostname");
```

The *addThread* and *moveThread* methods take explicit host names as an argument in order to select the processing node on which the thread should be created. DPS offers several helper objects that can be used for dynamically generating lists of host names at runtime. These objects are presented in the next section.

The threads in the thread collection are identified by unique thread identifiers that are used in the *removeThread* and *moveThread* methods. The usual technique for addressing threads in a thread collection is by their index (position) within the thread collection. However, this index does not necessarily stay constant over the duration of program execution, since threads might be removed, causing the indices of all following threads to be decremented. Therefore, two functions are provided in the thread collection to convert from identifiers to indices and vice versa.

```
// Convert from identifier to index
size_t threadIndex = processThreads.getThreadIndex(threadId);

// Convert from index to identifier
ThreadId threadId = processThreads.getThreadId(threadIndex);
```

3.5.4 Creating thread mapping strings

The thread mappings specified as input parameters to the *addThread* and *moveThread* methods are strings containing a list of host names. DPS provides two functions that generate such lists at runtime, which are useable in many common cases. The first function uses the provided configuration file and the command line processing abilities of the DPS library to recover the mapping string. The following source code shows how to make use of this approach:

```
processThreads.addThread(getController()->getConfig().getValue(
  "proc", "host1 host2 host3"));
```

The above line will use the value of the command line parameter *proc* as a mapping string. If the parameter is not specified on the command line, it will attempt to find the value in the DPS configuration file. As a final fallback, the set of hosts *"host1 host2 host3"* will be used. This approach for generating thread mapping strings is suitable for simple applications and for testing purposes.

The second method provided as part of the DPS library is the *pattern mapper*. The pattern mapper reads a list of nodes from a configuration files, and assembles these nodes into a mapping string according to an application provided pattern. The mapping pattern has the following format:

```
<nodes>[x<multiple>][+<offset>][b<backups>]
```

The initial element indicates the number of nodes to use. The multiple indicates how many times each node should be used. Therefore, the resulting size of the thread collection will be <nodes> multiplied by <multiple>. The offset indicates how many hosts should be skipped in the host file before starting the mapping. The offset can be used for example to create disjoint thread collections. The backup threads are used by the fault tolerance mechanism described in Chapter 6. The following source code shows how the pattern mapper is typically used, with both the name of the host file and the pattern being specified on the command line:

```
// Create a PatternMapper object
dps::PatternMapper pm
   (getController()->getConfig().getValue("map","nodes.map"));

// Using mappings generated by the PatternMapper
processThreads.addThread
   (pm.get(getController()->getConfig().getValue("pat","3")).c_str());
```

The name of the host file is specified in the command line parameter *map*, and the pattern is specified in the parameter *pat*. Let us consider a host file *nodes.map* containing three nodes *host1*, *host2*, and *host3*. Here are a few examples of the values returned by the pattern mapper:

```
// Three threads
pm.get("3")          returns: "host1 host2 host3"

// Four threads, two on each host, offset by one
pm.get("2x2+1")      returns: "host2 host2 host3 host3"
```

In typical applications, both the file containing the node list and the pattern will be generated by an external tool such as a scheduler or a cluster monitoring system. Such a tool, usable with DPS, is presented in Appendix B.

3.5.5 Flow graphs

Flow graphs are built in DPS by combining all their component elements (operations, routing functions, conditions, thread collections) using a *flow graph builder*. The C++ classes used for the operations, the loop conditions and the routing functions are inherently static, since they are compiled by the C++ compiler. On the other hand, the flow graph creation process is handled at runtime, and therefore provides great flexibility in the assembly of the various elements.

The flow graph builder is implemented in the *FlowgraphBuilder* object, which is used to combine sequences of operations into a single graph. The operations within the flow graph are described by a *FlowgraphNode* object. The *FlowgraphNode* object encapsulates the operation class, the thread collection within which the operation is to execute, and the routing function for selecting the target thread within the thread collection. A sequence of operations is created by combining several *FlowgraphNode*s with the >> (right shift) operator, and added to the *FlowgraphBuilder* with the += operator. These overloaded operators provide a natural syntax for describing sequences of operations.

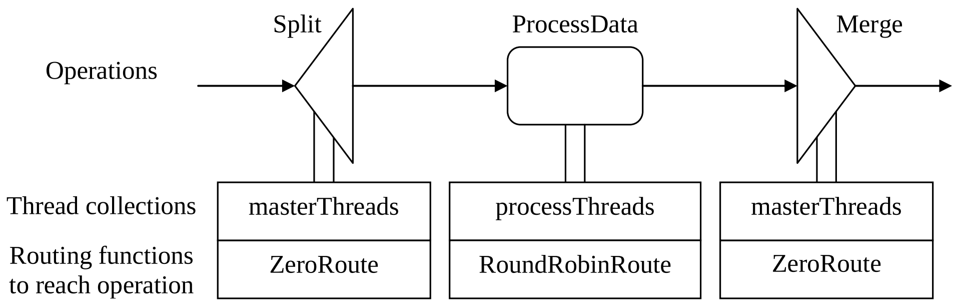


Figure 38. Simple flow graph for a compute farm, showing operations, thread collections, and routing functions

For example, to create the simple flow graph illustrated in Figure 38, the following source code would be used:

```
// Declare the flow graph nodes (operations)
dps::FlowgraphNode<Split, dps::ZeroRoute > split(masterThreads);
dps::FlowgraphNode<ProcessData, dps::RoundRobinRoute>
                                           process(processingThreads);
dps::FlowgraphNode<Merge, dps::ZeroRoute> merge(masterThreads);

// Create a flow graph builder object
dps::FlowgraphBuilder graphBuilder;

// Add a chain with the three operations
graphBuilder += split >> process >> merge;
```

The elements that are statically known at compilation time, such as the operation type and routing function type, are specified as template parameters to the flow graph node. These elements are used by DPS to perform some validation of the flow graph at compilation time to ensure that the flow graph is valid. DPS verifies the following points:

- The type of the threads in the thread collection specified for a *FlowgraphNode* is the type specified for the operation's execution thread. For operations that do not specify a specific thread type, any thread type is acceptable.
- The operation and the routing function specified for a *FlowgraphNode* can be created by the library using the class factory in order to ensure that DPS can create these objects on remote nodes that execute the application. The class factory is described in Chapter 5.
- The operation before the >> operator has at least one output data object type in common with input data object types of the following operation. This check ensures that the two operations can effectively communicate with one another.

In order to create complex flow graphs containing multiple branches, multiple sequences of operations can be added to the flow graph builder.

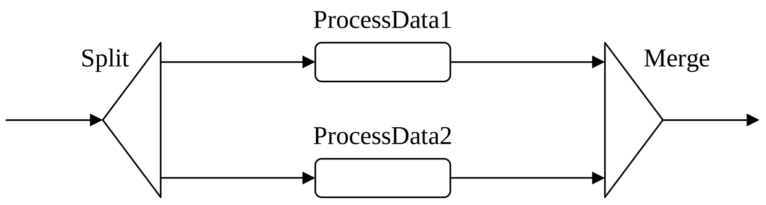


Figure 39. Flow graph with multiple branches

For example, to create the flow graph illustrated in Figure 39, the following source code would be used (assuming the same routing functions and thread collections as shown in Figure 38 are used):

```
// Create a flow graph builder object
dps::FlowgraphBuilder graphBuilder;

// Declare the flow graph nodes (operations)
dps::FlowgraphNode<Split, dps::ZeroRoute> split(masterThreads);
dps::FlowgraphNode<ProcessData1, dps::RoundRobinRoute>
                                          process1(processingThreads);
dps::FlowgraphNode<ProcessData2, dps::RoundRobinRoute>
                                          process2(processingThreads);
dps::FlowgraphNode<Merge, dps::ZeroRoute> merge(masterThreads);

// Add a sequence to form upper path
graphBuilder += split >> process1 >> merge;

// Add a sequence to form lower path
graphBuilder += split >> process2 >> merge;
```

Since the flow graph nodes *split* and *merge* are used twice, the flow graph builder knows that the operations appearing between them are on different paths between these two flow graph nodes. This process can be extended to an arbitrary number of branches within the flow graph.

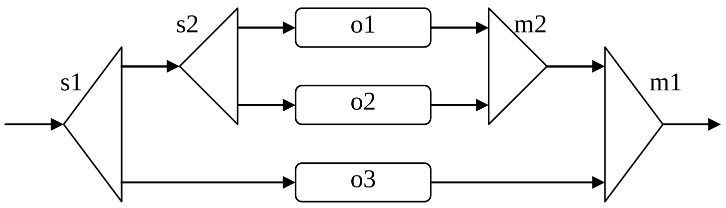


Figure 40.   Complex flow graph with multiple branches

The flow graph shown in Figure 40 can be created as follows (the declaration of the flow graph nodes has been omitted):

```
graphBuilder += s1 >> s2 >> o1 >> m2 >> m1;  // First branch
graphBuilder +=       s2 >> o2 >> m2      ;  // Branch with o2
graphBuilder += s1 >>       o3       >> m1;  // Branch with o3
```

Only the parts of the flow graph that have not yet been specified need to be added to the flow graph builder. In the above example, the path from *s1* to *s2* is only added once. Flow graphs of arbitrary complexity may be constructed by using this mechanism, since the various operations on the flow graph builder can be wrapped in any type of C++ control structures, such as loops or conditions, allowing the flow graph to be constructed at runtime to meet the application's requirements. Once the flow graph has been completely described within the flow graph builder, the final flow graph object is created by the controller.

```
dps::Flowgraph graph =
  getController()->createFlowgraph("graph",graphBuilder);
```

Once the controller owns the flow graph, it can no longer be modified. However, the same operations, routing functions and thread collections can be reused in any number of distinct flow graphs.

3.5.6 Operations

The nodes of a flow graph are its operations. Within DPS, all operations share a common syntax and programming model. DPS uses C++ classes to represent operations. In order to customize the functionality of an operation, the developer derives a custom class from a base class provided by the DPS library. The following base classes are provided, representing the four operation types supported by DPS: *LeafOperation*, *SplitOperation*, *MergeOperation* or *StreamOperation*. The base class is a template, taking as arguments several of the operation's attributes: the input data object type, the output data object type, and the thread type on which the operation executes. The overall syntax is therefore the following (example given for a leaf operation):

```
class UserLeafOperation : public dps::LeafOperation
  < InputDataType, OutputDataType, UserThreadType >
{
  void execute(InputDataType *in)
  {
    /* Perform some processing */
    postDataObject(new OutputDataType());
  }
  IDENTIFY(UserLeafOperation);
};
```

The template arguments are used for validating the sequence of operations within a flow graph at compile time. Like all classes used within the DPS framework, the IDENTIFY macro is used to provide basic type reflection features and enable runtime object instantiation within the DPS class factory.

The main body of the operation is located in the *execute* member function. This function is called by the DPS library when a data object needs to be processed by the operation. An operation can contain any amount of C++ code. When an operation wishes to post a new data object, it can call the *postDataObject* function provided by the base class. A leaf operation must post exactly one output data object.

When creating a split operation, any type of C++ looping constructs, such as *for* or *while* loops, can be used to create any number of output data objects. The following source code shows an example using a *for* loop.

```
class UserSplitOperation : public dps::SplitOperation
  < InputDataType, OutputDataType, UserThreadType >
{
  void execute(InputDataType *in)
  {
    // Create 10 output data objects
    for(int i=0;i<10;++i)
    {
      // Post a data object
      postDataObject(new OutputDataType());
    }
  }
  IDENTIFY(UserSplitOperation);
};
```

Merge operations are implemented in a similar fashion, with the main difference that they need to receive multiple input data objects. This is achieved by using the *waitForNextDataObject* function. This function returns the next data object that is to be processed by the merge, or *NULL* if all the data objects have already been processed. Therefore a typical merge operation might look as follows:

```
class UserMergeOperation : public dps::MergeOperation
  < InputDataType, OutputDataType, UserThreadType >
{
  void execute(InputDataType *in)
  {
    // do-while loop until waitForNextDataObject returns NULL
    do
    {
      /* Do something with data object in */
    }
    while((in = waitForNextDataObject())!=NULL);

    // All input data objects have been received,
    // post output data object
    postDataObject(new OutputDataType());
  }
  IDENTIFY(UserMergeOperation);
};
```

Like leaf operations, the merge operation must post exactly one data object. It must also loop on *waitForNextDataObject* until *NULL* is returned in order to ensure that no orphan data objects are left within the parallel schedule.

A stream operation uses the same general structure as a merge operation, but it may contain multiple calls to *postDataObject* like split operations. The following source code segment illustrates a typical stream operation:

```
class UserStreamOperation : public dps::StreamOperation
  < InputDataType, OutputDataType, UserThreadType>
{
  void execute(InputDataType *in)
  {
    // do-while loop until waitForNextDataObject returns NULL
    do
    {
      /* Do something with data object in */
      if( /* Some condition is met*/ )
      {
        // Post an output data object
        postDataObject(new OutputDataType());
      }
    }
    while((in = waitForNextDataObject())!=NULL);

    // Execution of stream operation can continue beyond the while
    // loop, in particular to post some more data objects
  }
  IDENTIFY(UserStreamOperation);
};
```

The DPS operation model was designed to be simple and orthogonal, providing the same syntax for all operation types. The various looping constructs rely on standard C++ loops. This creates interesting challenges for the DPS library, since mechanisms such as flow control might need to suspend operations within the sequential C++ code, and resume execution at a later time. The solutions are described in Chapter 5, which describes the internal functions of DPS.

The operation examples shown previously all have one input data type and one output data type. However, flow graphs may contain multiple branches, where the branch taken by a data object is based on its type, as shown in Figure 41.

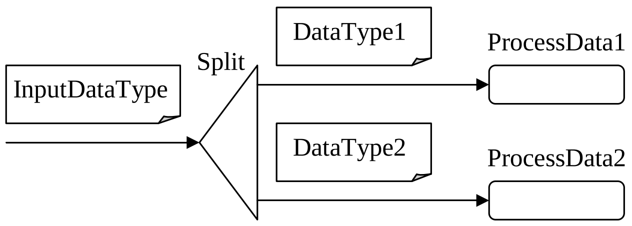


Figure 41. Flow graph with multiple branches

In this example, the *Split* operation can output both *DataType1* and *DataType2*, causing the execution of either *ProcessData1* or *ProcessData2* respectively. In these cases, all the valid data types are specified by using the *tv* template.

```
class Split : public dps::SplitOperation
  < InputDataType, dps::tv<DataType1,DataType2>, UserThreadType >
```

The *tv* template can take from 1 to 5 arguments indicating the allowed data types. DPS decides which path is taken on the graph by matching the output data types of an operation with the input data types of the following operations. DPS also verifies at compile time that multiple data types are only specified where appropriate (i.e. a leaf operation may only specify one input data type and one output data type).

3.5.7 Routing functions

The role of a routing function is to select a thread within a thread collection on which an operation is to be executed. DPS provides a collection of basic routing functions that can be used for common scenarios:

- *ZeroRoute* – This routing function always returns 0. It is typically used for routing to split and merge operations running within a thread collection containing a single thread that is mapped to a master node.
- *ConstantRoute<value>* – This routing function returns the constant passed as template argument. It has a similar use as *ZeroRoute*.
- *RoundRobinRoute* – This routing function automatically distributes the data objects in round-robin fashion to all the threads within the target thread collection. It is typically used to achieve a simple distribution on worker threads in embarrassingly simple parallel applications.
- *RandomRoute* – This routing function returns a random index within the target thread collection. It has a similar uses as *RoundRobinRoute*.
- *NoRoute* – This routing function returns the same index as the previous operation was run on. It is typically used when a data object should stay within the same processing node.
- *LoadBalancedRoute* – This routing function is used to request the automatic load balancing feature provided by the DPS library. Automatic load balancing should only be used for operations running on stateless threads.

For more complex data-dependent routing, the developer can provide his own routing functions. A custom routing function returns a thread index within the target thread collection. In order to compute this index, the routing function can access the input data object and the thread collection size. No further information is available, since the routing function is not attached to any particular thread. As with operations, the routing functions are expressed using a C++ class deriving from the base class *Route*.

```
class UserDefinedRoute : public dps::Route<InputDataType>
{
  size_t route(InputDataType *in)
  {
    return /* A thread index based on input data object */;
  }
  IDENTIFY(UserDefinedRoute);
};
```

The *Route* class takes as a template parameter the type of data object to be routed, which is used as parameter to the *route* method. The *route* method returns the index of the desired thread. The number of threads in the collection to which the data object is sent can be obtained by calling the *getThreadCount* member function of the *Route* class.

3.5.8 Flow graph loops

Flow graph loops are used in order to enable repeated execution of sequences of operations within parallel schedules. DPS implements flow graph loops by inserting special *loop* constructs into the flow graph. A loop construct is composed of a *condition* and a target operation. When a data object reaches a loop construct in a flow graph, the condition is evaluated. If the condition is true, execution in the flow graph continues at the target operation of the condition, otherwise execution continues on the next operation in the flow graph. The condition is implemented within a class deriving from the base class *Condition*. Like routing functions, the condition needs to take its decision based only on the content of the incoming data object. The structure of a condition is shown in the following source code sample:

```
class UserCondition : public dps::Condition <InputDataType>
{
public:
  bool condition(InputDataType *in)
  {
    return /* true or false */;
  }
  IDENTIFY(UserCondition);
};
```

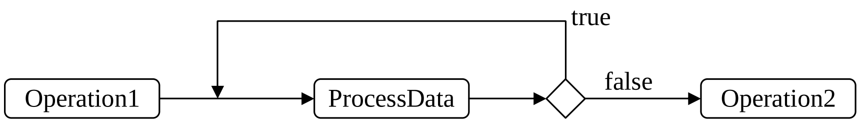


Figure 42. Flow graph segment containing a loop

Figure 42 illustrates a simple flow graph segment containing a loop. This segment can be created with the following source code fragment:

```
// Declare the flow graph nodes (operations)
dps::FlowgraphNode<ProcessData,ProcessRoute>
                                       processData(processingThreads);

// Declare the flow graph nodes (loops)
dps::FlowgraphLoop<LoopCondition> loop(processData);

// Add a chain to flow graph
graphBuilder += operation1 >> processData >> loop >> operation2;
```

The loop is inserted with a *FlowgraphLoop* element. The *FlowgraphLoop* is used in the same fashion as the *FlowgraphNode*, by inserting it into the flow graph builder with >> operators. The template parameter of the *FlowgraphLoop* is the condition to use, and its single argument is the target of the loop when the condition is true (in the above example the *ProcessData* operation). Loops can be used to create very long pipelined parallel sequences, as illustrated in the LU decomposition example presented in section 4.3.

3.5.9 Working with parallel components

Declaring a parallel component is identical to creating any other flow graph. The DPS library automatically exposes any registered flow graph to other applications. For example, let us reconsider the previous striped file system example illustrated in Figure 43.

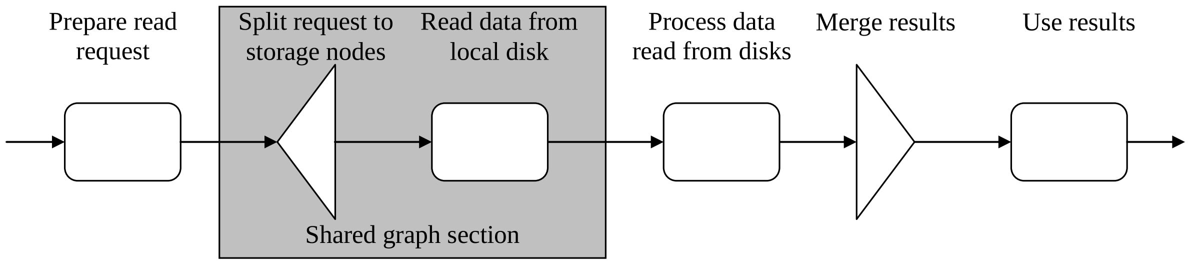

Prepare read request
Split request to storage nodes
Read data from local disk
Process data read from disks
Merge results
Use results
Shared graph section

Figure 43. Flow graph using a parallel service to read from a striped file

A typical mapping scenario for this application is illustrated in Figure 44. Both the client application and the striped file system are running on the participating nodes, and have individual network addresses. The client application needs to know the network address of an instance of the striped file system in order to be able to obtain the shared flow graph section.

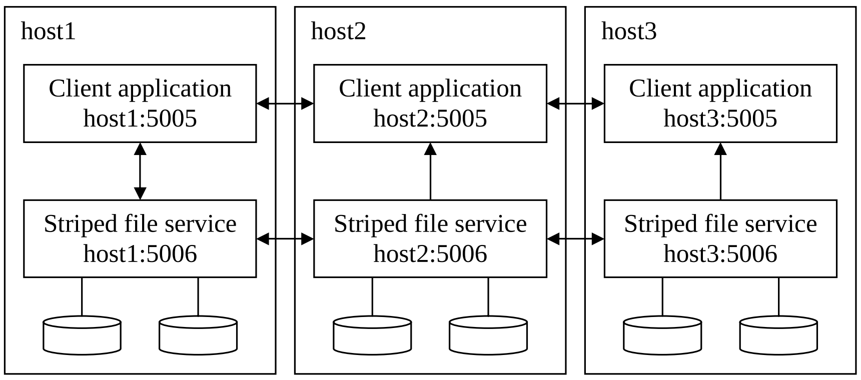

host1
Client application host1:5005
Striped file service host1:5006
host2
Client application host2:5005
Striped file service host2:5006
host3
Client application host3:5005
Striped file service host3:5006

Figure 44. Typical mapping for parallel components

The striped file system parallel component of Figure 43 is created with the following source code in the application that provides the parallel component. Only the flow graph construction code is shown.

```
dps::FlowgraphNode<SplitRequest,MainRoute> split(mainThreads);
dps::FlowgraphNode<ReadData,ReadRoute> read(diskThreads);
dps::FlowgraphBuilder readGraphBuilder = split >> read;

dps::Flowgraph readGraph =
        getController()->createFlowgraph("diskgraph",readGraphBuilder);
```

After the flow graph has been created, it is available to other applications as *"diskgraph"*. In order to integrate this flow graph, an application must first recover it, and then use it in its own flow graph building process. The flow graph is found by using the *findFlowgraph* method of the DPS controller, which takes as arguments the name of the flow graph to be found and the network address on which to request the flow graph. The following source code requests the parallel component flow graph, creating a local copy of the flow graph.

```
// Request the read graph from a host that is running the striped file
// service parallel component. readNetworkHost is the address of an
// instance of the striped file system component.
dps::Flowgraph diskGraph =
           getController()->findFlowgraph("diskgraph",readNetworkHost);
```

The parallel component flow graph is inserted into the local flow graph builder by packaging the parallel component flow graph within a *FlowgraphSection* object, which behaves like the usual *FlowgraphNode*. Since the flow graph is located within another application, no appropriately templated form of the operations within that flow graph are available. Therefore, the *FlowgraphSection* object needs to specify the input and output data object types (here *ReadRequestObject* and *DataBlockObject* respectively) for the flow graph section in order to enable the DPS compile-time type checking. The constructor argument is the flow graph to use. When the resulting flow graph is used to invoke a parallel schedule, DPS automatically takes care of transferring the data objects from one application to another. The following source code creates the complete flow graph shown in Figure 43 in the client application.

```
// Create a flow graph section corresponding to the remote flow graph
// The acceptable input and output data objects need to be specified
dps::FlowgraphSection<ReadRequestObject,DataBlockObject>
                                          remoteDiskSection(diskGraph);

// Create a flow graph containing the remote section
dps::FlowgraphNode<PrepareRequest,MainRoute> prepare(mainThreads);
dps::FlowgraphNode<ProcessData,dps::NoRoute> process(processThreads);
dps::FlowgraphNode<MergeResults,MainRoute> merge(mainThreads);
dps::FlowgraphNode<UseResults,MainRoute> use(mainThreads);

dps::FlowgraphBuilder theGraphBuilder =
  prepare >> remoteDiskSection >> process >> merge >> use;

dps::Flowgraph theGraph =
             getController()->createFlowgraph("graph",theGraphBuilder);
```

An important constraint on parallel components is on the routing function for the first operation after the imported graph segment. Since the data object is routed from within the context of the application that exports the flow graph segment, it needs to be defined within that application as well as within the client application that uses the flow graph segment. In the example, this constraint applies to the routing function used for *ProcessData*, which is a built-in DPS routing function (*NoRoute*), and therefore available within both the client and the server applications. The *NoRoute* routing function simply redirects the data objects to the threads with the same identifier. The data objects are therefore kept on the local node if both the *diskThreads* (the thread collection used for the disk read operations) and *processThreads* (the thread collection used for processing the data that was read from disk) thread collections have been mapped to the same nodes in the same order.

Parallel component flow graph sections may also be used within the application that defines them, for example to create modular flow graph designs.

3.5.10 Data objects

The data objects that are passed along the flow graph from one operation to another are also instances of user-defined C++ classes. Since these objects also need the capability to be passed from one address space to another, potentially over a network connection, they need to be *serializable*. Since C++ provides no built-in reflection and serialization capabilities, DPS provides several mechanisms to provide these missing features. Consider the following simple data object:

```
class SimpleDataObject
{
  int a;
  int b;
};
```

For such data objects, a simple memory copy is useable as a serialization mechanism within homogeneous environments (same platform, same compiler). This type of mechanism is available using DPS's *SimpleSerial* serializer. In order to transform our simple data object into a valid DPS data object, it needs to be derived from *SimpleSerial* to provide the serialization functions, and to contain the *IDENTIFY* macro to provide the class factory functions.

```
class SimpleDataObject : public dps::SimpleSerial
{
  int a;
  int b;

  IDENTIFY(SimpleDataObject);
};
```

This simple model is however very limited in scope, and does not exhibit a reliable behavior within heterogeneous environments. Therefore, a more general approach needs to be devised, provided by DPS's *AutoSerial* serializer. *AutoSerial* provides the ability to serialize many different data types, including STL containers, arrays and simple pointers. For example, consider the following complex data object:

```
class ComplexDataObject
{
  int a;                // An integer
  AnotherObject *obj;   // Pointer to an instance of AnotherObject
  std::vector<std::string> strings;  // A vector of strings
  int c[32];            // Fixed-size array of integers
};
```

This complex data object can be made serializable by deriving from DPS's *AutoSerial* and by adding several macros to the data declaration.

```
class ComplexDataObject : public dps::AutoSerial
{
  CLASSDEF(ComplexDataObject) // Declare class name
  MEMBERS                     // Declare class members
    ITEM(int, a)
    ITEM(SingleRef<AnotherObject>, obj)
    ITEM(std::vector<std::string>, strings)
    ARRAY(int,c,32)
  CLASSEND;
};
```

The advanced serialization mechanism wraps all the members into *ITEM* macros, with the exception of the array which needs an additional argument (the array size). The pointer also requires a special treatment in order to take care of object lifecycle management. The details of *AutoSerial* and object lifecycle management are presented in Chapter 5.

3.5.11 Executing parallel schedules

The previous sections described all the components that can be used for constructing thread collections and flow graphs. Once a flow graph has been created, it can be executed as a parallel schedule.

```
InputDataObject *in = new InputDataObject();
OutputDataObject *out = (OutputDataObject*)
                            getController()->callSchedule(theGraph,in);
```

The *callSchedule* method starts an instance of a parallel schedule, using the specified flow graph and input data object. When the execution of the schedule is complete, the resulting output data object is returned. It is also possible to invoke parallel schedules asynchronously, since they are executing within their own threads.

```
InputDataObject *in = new InputDataObject();
dps::ScheduleId scid = getController()->callScheduleAsync(theGraph,in);

// Do something else while the parallel schedule is running

OutputDataObject *out = (OutputDataObject*)
                                getController()->waitForSchedule(scid);
```

DPS allows multiple parallel schedules to be executed simultaneously based on the same flow graph and thread collections, as shown in the following example.

```
InputDataObject *in1 = new InputDataObject();
InputDataObject *in2 = new InputDataObject();
dps::ScheduleId scid1, scid2;

// Call schedule twice asynchronously with different input data objects
scid1 = getController()->callScheduleAsync(theGraph,in1);
scid2 = getController()->callScheduleAsync(theGraph,in2);

// Do something else while the parallel schedules are running

// Collect output data objects of both schedules
OutputDataObject *out1 = (OutputDataObject*)
  getController()->waitForSchedule(scid1);
OutputDataObject *out2 = (OutputDataObject*)
  getController()->waitForSchedule(scid2);
```

3.6 DPS runtime

DPS needs a flexible runtime environment, enabling applications to add and remove nodes from their working set at any time. These needs are addressed by a mechanism called *connectors*, which controls how new instances of applications are started on the participating nodes. DPS provides three basic connectors: *RSHConnector*, *KernelConnector*, and *LocalConnector*. The connectors are responsible for the addressing mechanism used for identifying nodes and executing new instances of the parallel application.

The *RSHConnector* is designed to use existing operating system services for starting remote applications, such as *rsh* or *ssh*. These remote shell utilities are commonly installed on clusters running Unix-based operating systems. However, startup times for remote shell sessions, in particular with *ssh*, can be fairly high, leading to long application startup times.

The *KernelConnector* uses a DPS-specific application launcher, called a *kernel*, which needs to be running on all nodes that are participating in the execution of the DPS application. The kernel provides faster startup times, and additional features such as resource monitoring.

Finally, the *LocalConnector* is used to start multiple instances of the same application on the local host. It is mainly used for debugging and testing purposes, which are simpler when a single node is used.

3.6.1 Networking

Internally, DPS uses an abstract networking provider, called the *network layer*. This layer abstracts from DPS the internal aspects of the underlying network, such as addressing and data transfer. Currently, DPS provides only a single network layer for TCP/IP networking. A future DPS extension will provide an additional network layer for a simulated network. The internal workings of the network layer are described in Chapter 5.

When using the TCP network protocol, individual instances of DPS applications are identified by the IP address of the host they are running on and the port number on which they are listening for incoming connections. When multiple instances of DPS applications are running on the same host, they must use distinct port numbers. The mechanism used for assigning the port number is dependent on the connector being used and the DPS library configuration.

When using the rsh connector, the port number must be specified on the command line or in a configuration file. When additional instances of an application are started on remote processing nodes, they listen on the same port number as the initial application instance. The user needs to ensure that no collisions in port numbering occur when multiple applications are sharing the same node.

When using the kernel connector, the port number allocation is handled by the kernels. All kernels listen on a fixed port specified on the command line or a configuration file. Port numbers are dynamically assigned to the applications as they are executed by the TCP/IP implementation, and the kernel keeps track of the running applications in order to ensure that they can be contacted by remote instances. Therefore using the kernel is ideal when multiple DPS applications are sharing the same cluster, since port number conflicts are avoided even when running multiple instances of the same application on the same nodes.

When using the local connector, multiple instances of an application are identified only by their port number, since they are all running on the same node. The port numbers are read by the local connector from the thread mapping string, and the new instances are launched with the appropriate port.

3.6.2 Debugging

Debugging parallel applications is usually a cumbersome task, since the application runs several instances of itself on multiple nodes. Since DPS provides a flexible thread mapping model, any application can be made to run within the boundaries of a single process with an identical layout as when running on multiple nodes. The complete parallel structure of the application is thus preserved, but the application can be handled by a conventional debugger. Such a configuration allows most of the structural bugs present in the application to be found. Further errors related to data transfers (serialization and deserialization of data objects) or particular synchronization issues can be discovered by using the *LocalConnector* to run multiple instances of the application in separate processes on a single node, with networking. These debugging options allow the majority of the bugs within an application to be discovered before actually deploying it on a cluster.

3.7 Comparison with CAP

The DPS implementation of the parallel schedules model differs from the previous CAP implementation [Gennart99] in numerous ways. The following paragraphs highlight some of the differences. For a brief description of CAP, see Appendix A. DPS is a 'clean room' implementation of parallel schedules. Beyond the most basic concepts such as the use of hierarchical split-process-merge flow graphs, no elements of CAP were reused in the implementation of DPS. Simple runtime concepts such as the approaches used for flow control and load balancing were also inherited from CAP.

DPS brings many new capabilities to the parallel schedules approach, in particular all the new dynamic features, allowing flow graphs and thread collections to be constructed and updated at runtime. The execution model present in CAP was static, similar to the one offered by MPI. The dynamic nature of DPS allows parallel schedules to gain important new characteristics, such as fault tolerance, parallel components and malleability.

Both CAP and DPS define applications by their flow graphs. The techniques used for defining the flow graphs are however completely different. CAP defines a set of predefined building blocks (constructs) that are combined to create a flow graph, whereas DPS provides a more flexible approach by combining sequences of operations. Whereas the CAP approach allows for simple application design, it cannot offer the flexibility found in the general parallel schedule model. In particular, developers have little control over split operations and merge operations, since most of their functionality is controlled by the construct that is used. The parallel CAP constructs provide a 'fill out the gaps' approach where some elements are fixed by the construct, in particular the looping model, and some other elements are filled out in various places in the application source code. Since DPS uses self-contained operations, all related code elements (such as loop statements and the contents of the loop in split operations) are located in the same piece of code and are formulated with the usual C++ syntax.

The declarations of constructs within CAP are based on extensions to the C++ language. CAP requires a complex preprocessor in order to parse CAP source code. Maintaining such a preprocessor and ensuring that it can process arbitrarily located CAP constructs within any type of C++ source code is an arduous and time-consuming task. The difficulty is further increased by having to ensure that the additional CAP constructs and keywords do not collide with the header files of other libraries. DPS is a simple C++ library, using conventional C++ techniques such as operator overloading and templates in order to provide the features required for constructing flow graphs. DPS therefore leaves C++ developers in a more familiar environment, and at the same time simplifies the maintenance of the framework.

3.8 Conclusion

This chapter provided an overview of the concepts and syntax of parallel schedules within the DPS framework. The concepts and syntax achieve the goals that were specified in Chapter 2. They provide a flexible, dynamic and orthogonal approach for specifying parallel schedules. Within DPS, developers can specify their split and merge operations by using the C++ constructs of their choice, within a single C++ function. The simple flow graph construction mechanism enables flexible constructs such as stream operations to be seamlessly integrated into the model.

Both flow graphs and thread collections are created at runtime in order to achieve maximum flexibility. The runtime definition allows applications to integrate parallel components exposed by separate running applications. The integration of parallel components is achieved by dynamically incorporating flow graph parts from several applications into a single flow graph. The thread collections can be altered at any time in order to dynamically adjust to changing runtime conditions, opening the way for the creation of malleable applications. The ability to update running thread collections also paves the way for fault tolerance.

Chapter 4

DPS Application Design

The DPS library provides a flexible environment for developing Parallel Applications. This chapter introduces several applications developed with DPS, and illustrates some of the design options offered by the parallel schedule approach. The performance of the various approaches is also studied.

4.1 Introduction

In this chapter, we focus on three applications implemented with the DPS framework. The first application is the game of life that was already presented in Chapter 2 as an illustration for parallel schedules. This application is of particular interest since many common parallel computation tasks use similar partitioning and communication patterns [Kumar93]. Examples include finite difference computations like thermal diffusion, or image processing tasks like skeletonization. Several implementations for this application are presented, and the performance implications of the various designs are analyzed. An extended version where the current state of the world can be queried using a flow graph exported as a parallel component is also presented.

The second application is a simple matrix block multiplication. This application is used to illustrate the benefits of overlapping communications and computations within the DPS library.

The third application is an implementation of the LU factorization, a classical linear algebra problem. This application is an example of a real-world problem, and illustrates the influence of deep pipelining in flow graph design and the importance of flow control.

4.1.1 Test platforms

Performance measurements are carried out for all of the applications presented in this chapter. These measurements were taken on two different clusters. The first cluster is composed of 4 dual-processor 733MHz Pentium III systems. These systems are equipped with 768Mbytes of memory, and run under Windows 2000. The systems are connected with both Fast Ethernet and Gigabit Ethernet, allowing both networks to be tested.

The second cluster is an ad-hoc collection of Sun Ultra 10 workstations which are accessible for interactive use. A set of idle machines with no users logged on is collected in order to dynamically create an "idle" cluster. The workstations are equipped with 350MHz UltraSparc processors, 256 Mbytes of memory, and run under Solaris 8. They are interconnected with a fast Ethernet network.

These two systems are used as test platforms for all the measurements presented throughout the rest of this document.

4.2 The game of life

A flow graph for implementing a parallel version of the game of life was presented in Chapter 2 in order to illustrate several parallel schedule concepts. This flow graph, shown again in Figure 45, proposes a very simple approach by clearly separating the initial border exchange and the subsequent computation of the next state. The two parts are separated by a global synchronization barrier in order to ensure that all participating nodes have completed the border exchange before performing the computation of the next state.

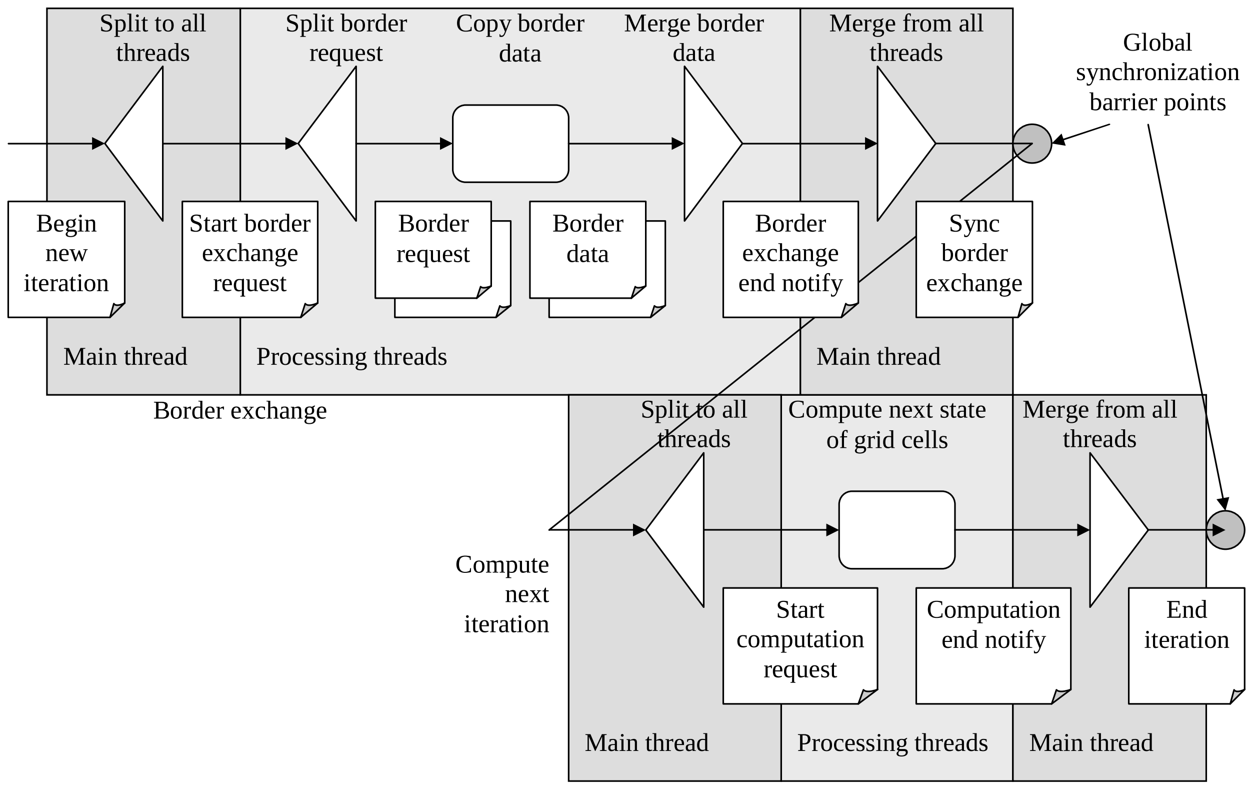


Figure 45. Complete flow graph for the game of life

The parallel schedule uses two thread collections: a collection with a single thread for executing all the global synchronization tasks (the main thread), and a collection distributed onto the participating nodes for performing all the computations (the processing thread). The processing threads store the current state of the world. At application startup, a simple split-operation-merge flow graph is used to distribute the initial state of the world to the processing threads.

4.2.1 Alternate flow graph designs

The previously presented flow graph is conceptually the simplest, but represents only one of many different possible approaches. The first aspect that can be improved in the flow graph is the removal of one of the two global synchronization points. The role of the synchronization after the border exchange is to ensure that all nodes have a consistent state of their borders. If the borders are stored in two buffers, one storing the current state and another being used for storing the new state, the border exchange operation will always copy the correct border when the neighbor requests it, even when computation of the new state is already complete. Using such a buffering mechanism allows the flow graph to be redesigned as shown in Figure 46a. However, this flow graph has a major shortcoming when it is being executed on multiple nodes: in some circumstances, the computation may have started on a node before that node has sent its borders to one of its neighbors. Since leaf operations are never interrupted, the neighbor will have to wait until the computation is completed before the border copy operation is executed locally. The returned border will be correct, since it comes from a separate buffer, but it will be delayed by the computation time. This additional delay causes the execution time of the parallel schedule for the computation of a single iteration to double, since two next state computations are performed sequentially rather than in parallel.

A possibility to avoid this phenomenon would be to exchange the computation and border exchange parts of the graph, thereby redefining the border exchange as preparation for the next iteration rather than for the current iteration. Since computation usually takes far more time than the border exchange, the appearance of the abovementioned inversion of the execution of the border exchange and the next state computation is much more unlikely, yet not totally impossible. The corresponding flow graph is shown in

Figure 46b. When using this flow graph, it is necessary to ensure that the borders are also initialized during the initial world distribution.

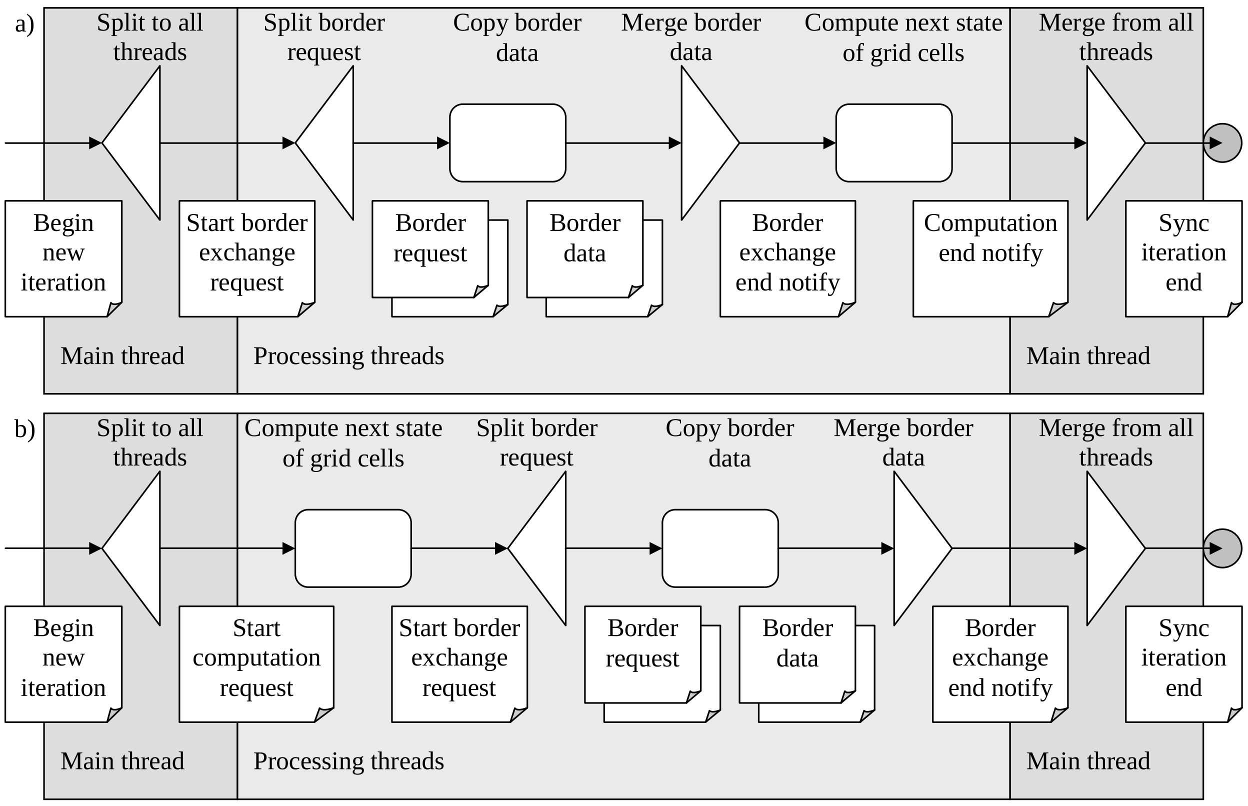


Figure 46. Reformed flow graphs for the game of life, using a single global barrier. a) exchange the borders first, b) compute the next state first

It is possible to further optimize the flow graph by taking into account the neighborhood requirements for the computation of the next state: most cells do not need the result of the border exchange in order to compute their next state. Only the cells immediately adjacent to the borders need the cells from the neighboring nodes. The flow graph can thus be optimized with an additional branch to compute the future state of the interior cells in parallel with the border exchange as shown in Figure 47. An additional advantage of this flow graph design is to eliminate the risk of the unintended execution order of operations that is inherent in the previous designs.

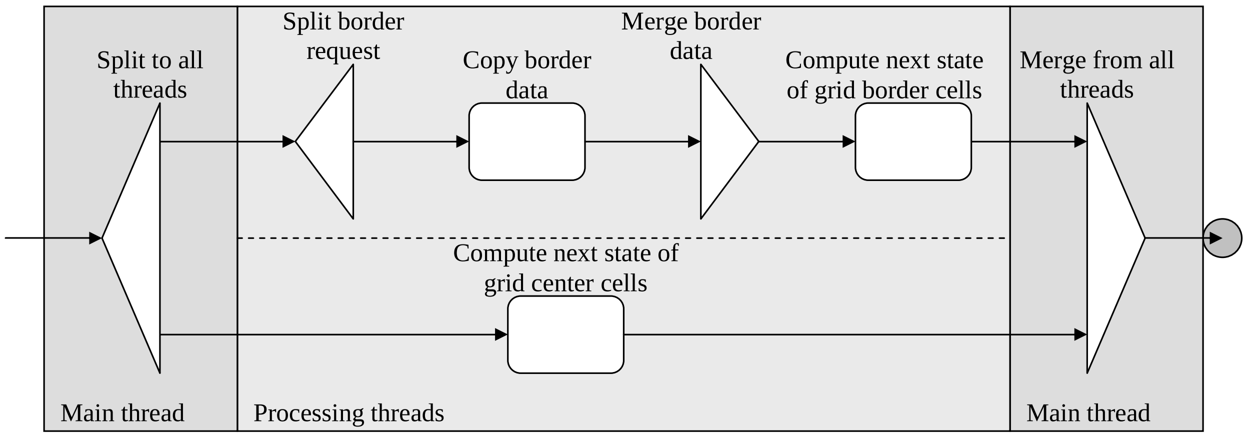


Figure 47. Optimized flow graph for the game of life

Both the upper and lower branch of the flow graph will require access to the same DPS thread, since they are working on the same data. In order to allow both branches to execute simultaneously, they need to be on two distinct operating system threads. Therefore, the thread collection used for storing the grid parts needs to be created with two operating system threads per DPS thread.

This flow graph configuration can reach optimal performance since DPS automatically overlaps computations with communications, allowing the computation of the center cells to proceed while the communications for the border exchange are taking place.

The implementation for all these four flow graphs is nearly identical, with only some minor changes required in some operations. The only major differences in the source code are in the declaration of the flow graphs, where the same operations are combined in different sequences. This ease of testing various parallel layouts makes DPS ideal for rapidly prototyping parallel applications.

4.2.2 Performance

The previous paragraphs presented four slightly different implementations for a parallel game of life. Since the computation code in all four variants is identical, any performance differences can be attributed to the parallel execution efficiency of the application.

Figure 48 shows the best-case execution time for a single iteration of the game of life for all four flow graphs. The application was run on the dual-processor Pentium III systems, as this cluster allows both Fast Ethernet and Gigabit Ethernet to be tested. Each processing node hosted two threads with aggregation disabled in order to ensure that both processors are used, for a total of 8 threads. This first series of measurements was performed by using a very small world size of 500x500 cells in order to highlight the internal overheads and latencies. The threads are mapped using a simple round robin distribution on the nodes, ensuring that two neighboring threads never share a node. This layout maximizes the communications occurring in the application. The execution time for the application running on a single dual-processor node with two threads is approximately 32 ms.

	Execution time [μs]	
Flow graph	Fast Ethernet	Gigabit Ethernet
Separate border exchange (Figure 45)	18830	19049
No sync, exchange first (Figure 46a)	16059	15794
No sync, compute first (Figure 46 b)	16103	15808
Overlap computation and exchange (Figure 47)	17936	17499

Figure 48. Execution times of successive iterations of the game of life running on 4 nodes

As expected, the first simple flow graph yields the lowest performance. The two intermediate flow graphs with the global synchronization removed have the fastest best-case execution times; however the previously mentioned execution order inversion appears periodically. The flow graph with the overlapped border exchange and computation yields intermediate performance, since the overhead of operating system thread switching plays a non-negligible role in these very short total execution times.

The appearance of the sequencing flaw where the inversion of the border exchange and computation happens on some nodes is highly dependent on the thread scheduling of the operating system and the network latency. Figure 49 shows the percentage of iterations on a 100 iteration run of the game of life where the inversion took place. In order to illustrate the incidence of thread scheduling, runs were also performed with a single thread per node, allowing the operating system to use both processors to schedule the internal DPS threads.

	Percentage of iterations exhibiting the execution order inversion (slow execution)			
	Fast Ethernet		Gigabit Ethernet	
Flow graph	4 threads	8 threads	4 threads	8 threads
No sync, exchange first (Figure 46 a)	3 %	20 %	0 %	18 %
No sync, compute first (Figure 46 b)	0 %	9 %	0 %	2 %

Figure 49. Percentage of iterations exhibiting the sequencing flaw when running on 4 nodes

The occurrence of the execution order inversion can be clearly shown to be caused by the latencies in the system. The execution is the most efficient on the lower-latency Gigabit Ethernet with a single thread per node, since only one processor is needed for the computations, and the other processor is available for all other DPS tasks such as network transfers, resulting in less operating system thread switching during execution.

The flow graphs without the global synchronization are a typical example of a flaw that can occur in the design of flow graphs when the asynchronous nature of the execution of parallel schedules is not taken into account. The resulting parallel schedule execution systematically produces the correct result, since there is no race condition or other serious error present in the source code, yet the execution time is not constant. For the following measurements, only the two flow graphs (Figure 45 and Figure 47) that do not exhibit this flaw are used.

The previous measurements were performed with a world size that was specifically selected to highlight the scheduling details of the various parallel schedule executions. Since the total overhead time caused by the flow graph design is independent of the world size, the relative differences between the various flow graphs diminish when a larger world size is used. The corresponding measurements are shown in Figure 50.

	Execution time [ms]		
Flow graph	500x500	2500x2500	5000x5000
Separate border exchange (Figure 45)	19.0	208	801
Overlap computation and exchange (Figure 47)	17.5	208	799

Figure 50. Execution times of successive iterations of the game of life running on 4 nodes

The optimized flow graph which overlaps computations and communications yields nearly identical performance on these runs of the game of life. Since only the immediate neighbors are required, the computation of states within the game of life however requires only a very compact neighborhood to be known. This is however not the case for all neighborhood dependent computations, and these situations can be simulated by artificially increasing the amount of border information that needs to be transferred. Figure 51 shows the resulting iteration execution times when the amount of border data is increased. For the flow graph that uses a global synchronization after the border exchange, execution time linearly increases with the amount of data that is added to the exchanged border information. The flow graph with overlapping communications and computations shows a lower slope. The initial slope is nearly identical for Fast Ethernet and Gigabit Ethernet, since the increase in execution time essentially reflects the additional processor usage caused by the communications using TCP/IP. For Fast Ethernet, an inflection point is reached in the curve when the communication starts taking more time than the computation, and the communication time becomes the limiting factor for the duration of iterations.

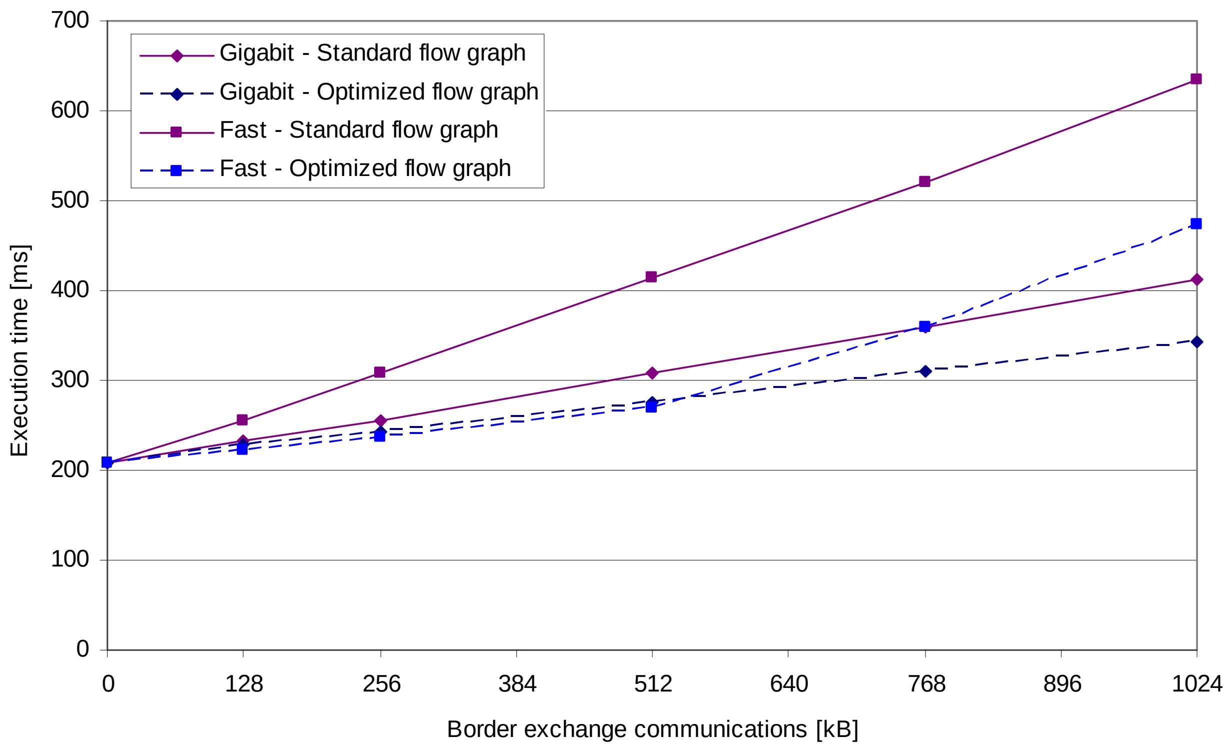


Figure 51. Execution time for a single iteration with varying border exchange data sizes

All the previous measurements were designed to illustrate the behavior of parallel schedules rather than achieve optimal performance. The following series of measurements illustrate the performance of the application on a varying number of nodes. The size of the world is directly proportional to the number of nodes that is used. Since the computation time is proportional to the size of the world, the execution time is expected to stay constant.

A first series of measurements, shown in Figure 52, is the parallel efficiency of the game of life running on the cluster of Pentium III systems, using from one to four nodes (2 to 8 threads). The threads on the dual processor nodes are arranged in order to minimize communications, i.e. the two threads on each node store neighboring world parts. The optimized flow graph with overlapping communications and computations yields a better efficiency when the number of nodes increases. However, on a single node, its efficiency is lower than that of the simple graph due to the overhead of scheduling the additional threads. The initial world size is set to 2000 by 2000 cells, and in order to keep a constant load on each node the world size is multiplied by the number of nodes in the tests.

A second series of measurements was performed on the ad-hoc cluster of Sun Ultra 10 Workstations. The results are presented in Figure 53. Both flow graphs show similar performance, with a very slight advantage to the optimized flow graph as the number of nodes increases. The initial world size is set to 4000 by 500 cells on a single node, and the world size is multiplied by the number of nodes in the tests, up to 4000 by 24000 cells on 48 nodes.

A final series of measurements was performed on the Sun workstations, to show the speedup of the application with a constant world size. The results are presented in Figure 54. The world size is set to 6000 by 6000 cells, and kept constant as the number of machines increases. The initial iteration time is 15 seconds on a single node, down to 737 milliseconds when running on 48 nodes. In this situation where the communication to computation ratio increases with the number of nodes, the optimized graph yields a much better performance on large numbers of nodes.

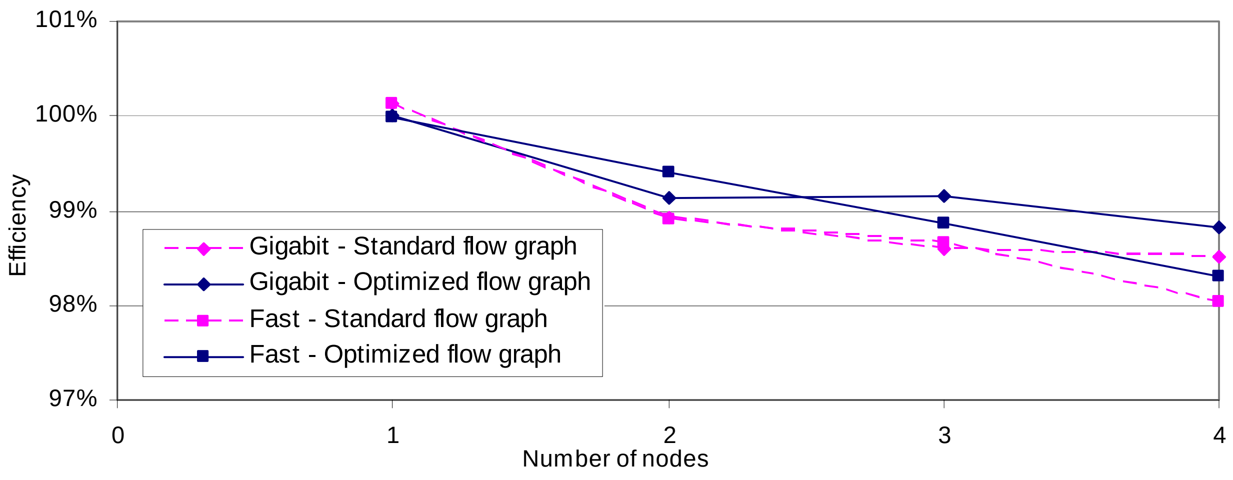


Figure 52. Efficiency of the game of life on Pentium III cluster (constant load per node)

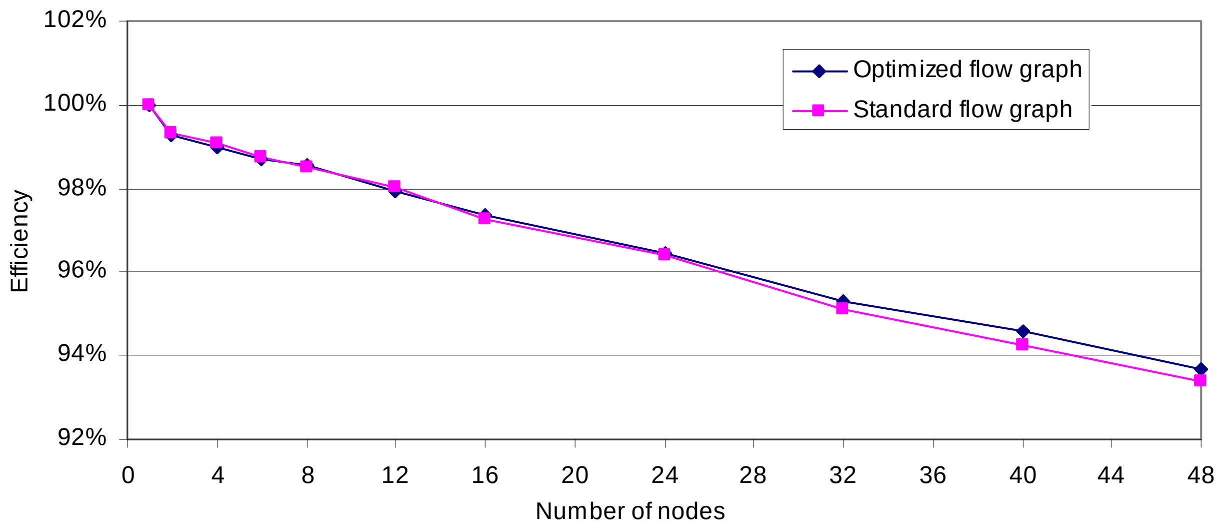


Figure 53. Efficiency of the game of life on Sun Ultra 10 cluster (constant load per node)

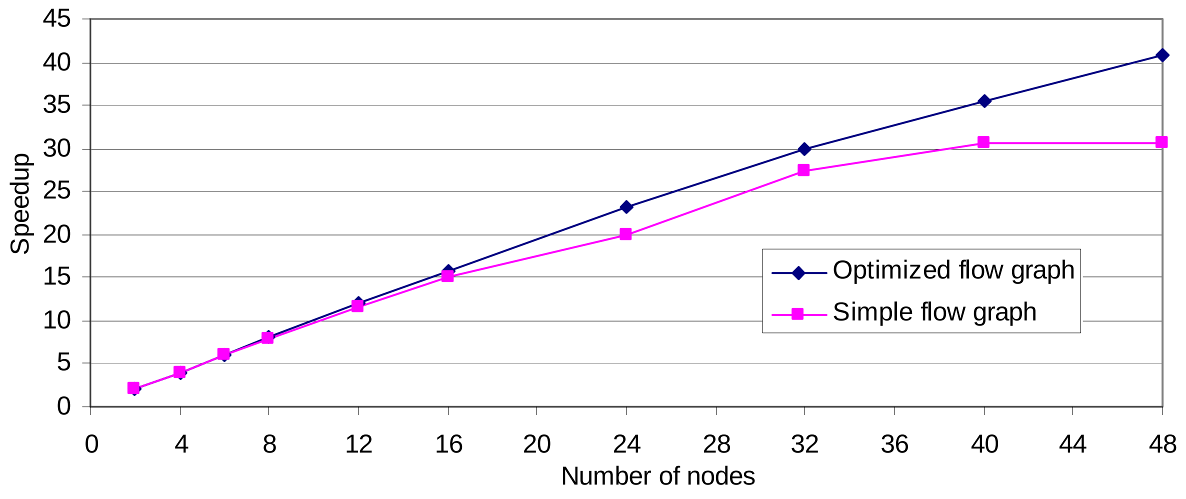


Figure 54. Speedup of the game of life on Sun Ultra 10 cluster (constant total load)

4.2.3 The game of life as a parallel component

In order to illustrate the parallel component features of DPS, the game of life was modified in order to include a flow graph segment that allows another application to query the current state of the computation. The additional flow graph segment is a simple split-operation-merge construct that can read part of the local state to an output data object. The reading operation is mapped onto the processing thread collection in order to ensure that it can access the current local state. The split and merge operations are mapped to a separate thread collection for handling the requests. The exported flow graph segment is illustrated in Figure 55. A typical application scenario with multiple clients simultaneously accessing the exported parallel component is shown in Figure 56.

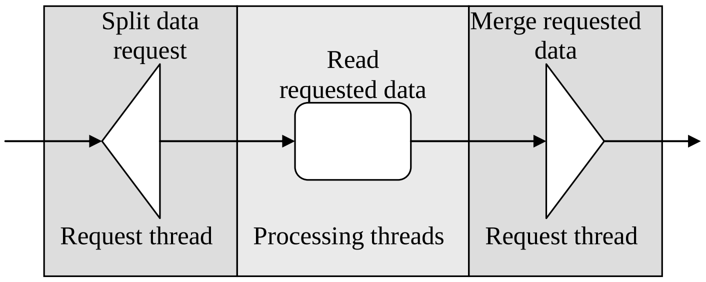


Figure 55. Flow graph segment for recovering part of the local world state in the game of life

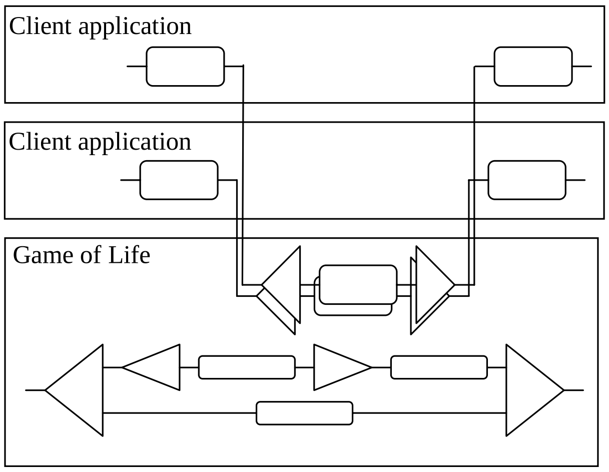


Figure 56. Multiple clients simultaneously accessing the game of life component

Since the read operation is mapped onto the processing threads, its execution will be interleaved with the computation and border exchange operations. If the game of life is using one of the simple flow graphs that use a single operating system thread per DPS thread, the read operation will always return a consistent state of the world, since it will never be running concurrently with a computation. However, when the game of life is using the optimized flow graph, this guarantee no longer applies. If a consistent state is critical, conventional synchronization primitives need to be used in order to ensure that both operating system threads are stopped when the copy is performed in response to a read request. If the read operation is mapped to the same operating system thread as the border exchange and computation operations, this synchronization can be obtained by mutually excluding the read operation and the center computation operation within an operating-system provided critical section.

In addition, there is no assurance that the various copies will get interleaved within the same iteration. Therefore, the various blocks that are collected by the merge operation of the collection graph might not be consistent with one another. The only solution to ensure that all the blocks are from the same iteration is to completely stop the computation on all nodes before collecting the blocks. The only means to achieve this is to mutually exclude the complete iteration flow graph and the query flow graph section, thereby ensuring that only one of the graph sections is running at a time. The mutual exclusion can be achieved by entering a critical section in the initial split operation of the iteration flow graph and leaving it in the corresponding merge operation. The same critical section is entered in the split operation of the query flow graph and left in the corresponding merge. Since both split-merge operation pairs are running on separate threads, no deadlocks can occur.

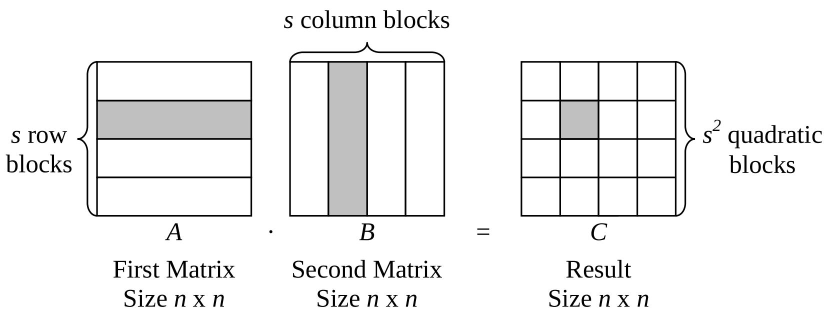
In order to measure the overhead of the component graph calls, we have set up a test case with the game of life continuously running with a world size of 5620x5620 cells. When running on 4 nodes of the Pentium III cluster without any calls to the query flow graph, each iteration takes 1000 ms. A single client application continuously requests randomly located fixed-sized blocks. Figure 57 shows the impact of the graph calls on the game of life execution speed. The call time comprises processing time (reading the world data from memory) and communication time. The implicit overlap of communications and computations enables flow graph calls to be executed very efficiently.

Query block size		Time per call (median)	Simulation iteration time	Average number of calls/sec	Total call time
width	height				
		No calls	1000 ms	None	None
40	40	1.66 ms	1041 ms	66.8	111 ms
400	400	22.14 ms	1284 ms	31.8	704 ms
400	2400	130.43 ms	1381 ms	6.9	897 ms

Figure 57. Game of life iteration time with and without parallel component calls

4.3 Matrix block multiplication

Matrix block multiplication is a simple technique for computing matrix multiplications in parallel. In order to multiply two matrices, the first matrix is split into horizontal bands, whereas the second matrix is split into vertical bands. These bands are then sent in pairs to the compute nodes which perform the multiplications and yield the blocks of the resulting matrix:



While this is not a very efficient way to perform matrix multiplications, it can serve as an interesting example for illustrating DPS's ability to overlap communications and computations. Assuming that the $n \times n$ matrix is split into s row or column blocks, the amount of communication is proportional to $(n^2/s)\cdot 2\cdot s^2+n^2 = n^2\cdot(2s+1)$, with (n^2/s) being the size of a column or row block, and s^2 being the number of matrix block multiplications. The number of computations is proportional to n^3. By keeping the size of the matrix n constant and varying the splitting factor s, the ratio between communication time and computation time can be modified. For this test, two 1024x1024 element matrices are multiplied on 1 to 4 compute nodes, with block sizes ranging from 256x256 (s = 4) to 32x32 (s = 32). This enables testing situations where either communications (s = 16 and s = 32) or computations (s = 4 and s = 8) are the bottleneck. The reductions in execution time due to overlapping of communications and computations and the corresponding ratios of communication time over computation time are given in Figure 58.

Block size	256		128		64		32	
Splitting factor	4		8		16		32	
Nodes	reduct.	ratio						
1	6.7%	0.22	9.1 %	0.45	17.6%	0.94	25.2%	2.09
2	13.6%	0.33	19.8%	0.66	28.7%	1.28	24.9%	2.76
3	15.8%	0.44	29.5%	0.97	32.1%	1.92	19.5%	4.19
4	23.9%	0.63	35.6%	1.36	27.2%	2.54	15.6%	5.54

Figure 58. Reduction in execution time of a 1024x1024 matrix multiplication due to overlapping and corresponding ratio of communication over computation time

The potential reduction *g* in execution time due to pipelining is either

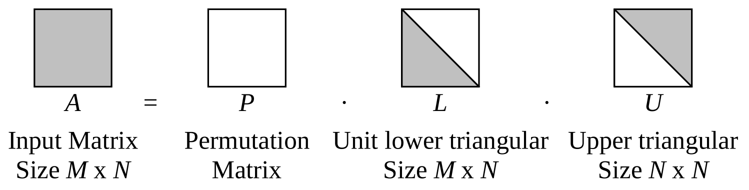
g = ratio / (ratio+1) if ratio 1, or

g = 1 / (1+ratio) if ratio 1.

Potential and measured reductions in execution time are the closest when the communication over computation time ratio is high, i.e. higher than 90%. This is easily explained by the fact that when communication dominates, processors tend to be partially idle. The highest gains in execution time are obtained at ratios of communication over computation times between 0.9 and 2.5. Out of a theoretical maximum of 50%, Figure 58 shows that DPS automatic pipelining yields execution time reductions between 25% and 35% when communication time is similar or up to 2.5 times higher than computation time. Total overlapping cannot be obtained since communications still require some processing time for the serialization and network protocol overheads.

4.4 LU decomposition

The LU decomposition [Golub96] is a particularly efficient technique for solving the large linear equation systems that are used in many common computation tasks. The LU decomposition is used to decompose a matrix *A* of size M x N, into three matrices *P*, *L*, and *U* such as:



The LU decomposition has been used as a benchmark for computer performance since 1979, when performance numbers appeared in the LINPACK users guide [Dongarra79]. It is currently still used for performance comparisons in the Top500 supercomputer list [Dongarra01]. The LU decomposition with partial pivoting is of particular interest due to the complex data dependencies involved in the computation, allowing testing of both floating point performance and communications performance.

4.4.1 Algorithm overview

The most common approaches to the parallelization of the LU decomposition are block-based, since these provide good data locality, which is necessary for efficient execution and parallelization. Let us examine the steps required to compute a right-looking block-based LU decomposition on a square matrix *A* of size *n* x *n*. Initially, the matrix *A* is split into four blocks

$$A = \begin{array}{cc} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} & \begin{array}{l} r \\ n-r \end{array} \\ \begin{array}{cc} r & n\text{-}r \end{array} & \end{array}$$ where A_{11} is a square block of size r x r.

We want to decompose this matrix as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = P \cdot \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

Step 1. The first step is to compute the matrices L_{11} and U_{11}, where L_{11} is a lower triangular matrix and U_{11} is an upper triangular matrix. These two matrices can be computed by Gaussian elimination, by using the rows from the lower part of the matrix (A_{21}) for pivoting as required.

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = P \cdot \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \cdot U_{11}$$

The permutations performed for pivoting are stored in the permutation matrix P[1]. All the computations and permutations can be performed in place on the matrix A, since the elements of L_{11} and U_{11} do not overlap. After this first step, L_{11}, L_{21} and U_{11} are known.

Step 2. In the second step, we compute U_{12} by solving the following triangular system:

$$A_{12} = L_{11} \cdot U_{12} \Rightarrow U_{12} = (L_{11})^{-1} \cdot A_{12}$$

Finally, to complete the LU decomposition, the matrices L_{22} and U_{22} should be lower and upper triangular respectively. This can be obtained by defining

$$\tilde{A}_{22} = L_{22} \cdot U_{22} = A_{22} - L_{21} \cdot U_{12}$$

Step 3. In a third step, $\tilde{A}_{22}$ is computed by multiplying L_{21} and U_{12} and subtracting the result from A_{22}. The previous steps can then be applied successively to $\tilde{A}_{22}$ until the complete matrix A has been decomposed.

Figure 59 illustrates the result of a single iteration of the algorithm.

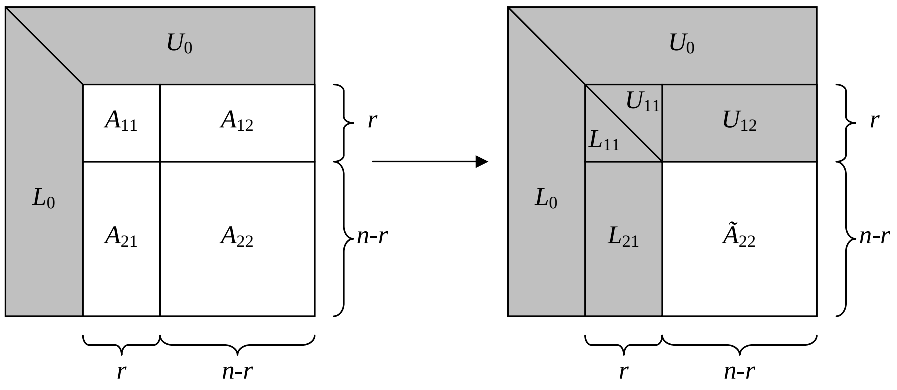


Figure 59. Result of one step of the block LU decomposition, the gray area represents the matrix parts that have already been decomposed

4.4.2 Parallel LU decomposition

Numerous approaches have been proposed for the parallel computation of the LU decomposition [Dongarra01][Toledo97][Choi96][Desprez95]. We have implemented several block-based LU decompositions with DPS in order to illustrate how such a problem can be mapped onto a parallel schedule.

When implementing a parallel LU decomposition, the matrix to be factorized needs to be distributed onto the computation nodes. The chosen distribution is critical to the performance and scalability of the application, since the algorithm does not use all parts of the matrix equally. The blocks in the lower right

[1] The permutation matrix P is a simple row permutation of the identity matrix. It is therefore usually stored as a vector of integers that indicate which rows were permuted [Press90].

corner are used significantly more than those in the top left corner. A classical distribution for the matrix is the 2-dimensional block cyclic distribution, where the computation nodes are arranged onto a rectangular grid of size P by Q, and this grid is subsequently tiled over the matrix blocks. Figure 60 illustrates such a distribution. The advantage of this distribution is the good load balance that is achieved during computation, since all nodes have some blocks of the lower right matrix corner.

Since all the elements of a column of the matrix are not located on the same computation node, the selection of the pivoting rows during the decomposition of A_{11} requires communications for every column in the matrix. In order to avoid these communications, a simpler 1-dimensional cyclic distribution can be used where complete columns of blocks are assigned to the individual computation nodes. This distribution greatly simplifies the pivoting, at the cost of a greater load imbalance within the final stages of computation.

Our initial implementation of the LU decomposition is intended to illustrate as many DPS features as possible, while keeping the implementation simple. We have therefore chosen a column-cyclic data distribution. For the high-performance implementation, we use a block-cyclic distribution.

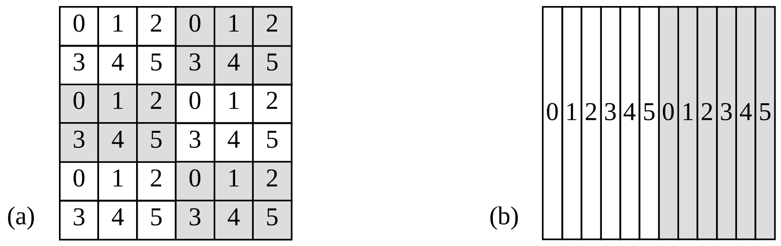


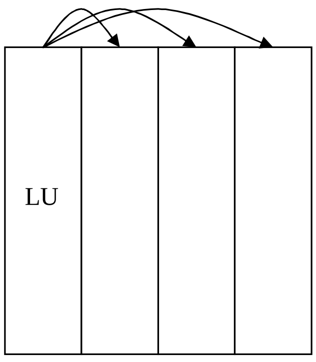
Figure 60. (a) 2-dimensional block cyclic distribution for a matrix divided into 6x6 blocks, mapped onto 6 nodes arranged on a 3x2 grid and (b) 1-dimensional cyclic distribution for a matrix divided into 12 columns of blocks, mapped onto 6 nodes

4.4.3 LU decomposition flow graph design

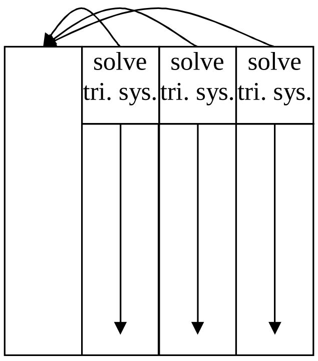
Our first design for the LU decomposition flow graph is based on a direct transposition of the sequence of processing tasks required for the block-based LU decomposition into a DPS flow graph. By using a column-cyclic data distribution, we avoid the need to communicate with other nodes in order to decide which rows are to be pivoted, thereby minimizing the number of parallel operations required in the implementation.

The LU decomposition is performed in three steps that are repeated for every column block of the matrix. The following paragraphs define the sequence of processing tasks on the matrix blocks involved in the LU decomposition. We also need to specify the corresponding communications in order to ensure that the required data is always locally available.

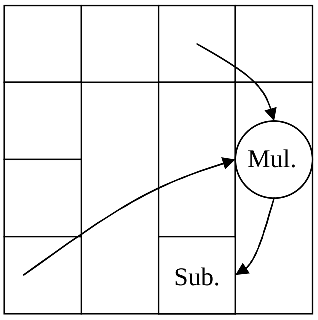
Step 1. Perform serially the Gaussian elimination on the node holding the first column of blocks. Send the resulting matrix L_{11} and the permutation matrix P to all other nodes.



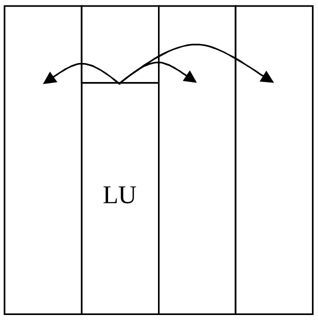
Step 2. On all other participating nodes, solve in parallel the triangular system $A_{12} = L_{11} \cdot U_{12}$. Solving such systems is the achieved by using the *trsm* function of BLAS. Perform the permutations specified by the received permutation matrix. Return the resulting part of U_{12} when the processing is complete.



Step 3. The initiating node now has L_{21} and a complete copy of U_{12}. It sends out multiplication requests to other nodes that compute the corresponding subparts of $L_{21} \cdot U_{12}$. The multiplications are not necessarily performed on the node where the result is needed, allowing to improve the load-balancing of the application. The additional communication cost is partly absorbed by the ability to overlap communications and computations. The results of the multiplication are sent out to the node where the subtraction from A_{22} is performed.



Step 1 (next iteration). As soon as the results of all multiplications required for the next column of computations have arrived, the process can restart with the Gaussian elimination. There is no need to wait for the completion of the multiplications for the following columns, since there is no dependency before step 2. The permutation matrix *P* is sent to the preceding columns (those for which processing is complete) in order to apply the permutations.



Based on this sequence of operations, the flow graph shown in Figure 61 was designed. This flow graph makes extensive use of stream operations in order to maximize pipelining within the application. A loop is used to repeat the operations for every column of blocks in the matrix.

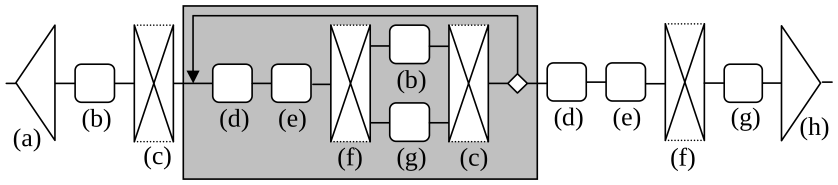


Figure 61. Flow graph for LU decomposition. The gray part is repeated for every column of blocks in the matrix by using a loop construct.

The operations within the flow graph perform the following tasks:

a) Perform Gaussian elimination for the top left block A_{11}, and send out triangular system solve requests to other columns (step 1).

b) Perform in parallel row flipping according to permutation matrix *P* and solve the triangular system in order to compute U_{12} for all other column blocks. Return the resulting matrix U_{12} (step 2).

c) Stream out multiplication requests as each column completes the triangular system solve. Each request contains a part of L_{21} and a part of U_{12}.

d) Parallel block-based matrix multiplication to compute $L_{21} \cdot U_{12}$. Return the result of the multiplication (first part of step 3).

e) Subtract the result of the multiplication from A_{22} in parallel, and send a notification that the multiplication is complete (second part of step 3).

f) As soon as the multiplication for the first column of blocks is complete, perform the next level Gaussian elimination, and stream out triangular system solve requests as soon as the multiplications for the other column blocks complete (back to step 1).

g) Perform row flipping on previous column blocks according to permutation matrix P.

h) Wait for the final row exchanges to complete before leaving applications.

The LU decomposition uses two thread collections for its operations. Most of the operations are executed within the *StorageThreads* thread collection. These threads store the matrix blocks within their local state. The multiplication operations are executed within a distinct thread collection *MultiplyThreads* in order to spread out the computation load onto multiple nodes. Figure 62 shows the mapping of operations to thread collections in the LU decomposition application.

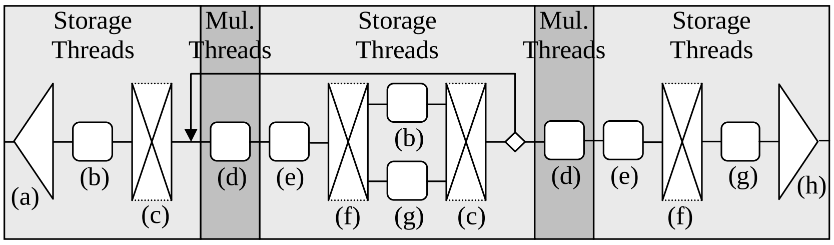


Figure 62. Flow graph for LU decomposition. The multiplication operation (d) is executed within its own thread collection.

4.4.4 Performance of the simple implementation

In order to keep the implementation of the LU decomposition simple, we have not used any external linear algebra libraries in our implementation. Therefore the overall performance of the application is low. However, many of the parameters that affect the performance of the LU decomposition can already be studied on this simple implementation: the influence of the chosen block size, which also affects the communication to computation ratio and the depth of the pipeline, the influence of pipelining, and the influence of flow control.

The chosen block size has a major influence on the efficiency of the LU decomposition. A small block size will yield a very long pipeline, and many very small parallel multiplications. These multiplications will have a high communication to computation ratio, and are therefore relatively inefficient. A large block size yields a very short pipeline, and a few large parallel multiplications. These multiplications have a low communication to computation ratio, and are more efficient. However, since there are fewer multiplications, the potential for parallel execution is more limited. The shorter pipeline composed of longer operations also leads to reduced processing power utilization as the execution nears the end of the pipeline. Figure 63 shows the execution times for an LU decomposition of a 2048x2048 matrix running on 8 Sun Ultra 10 workstations. The block size was varied from 32 to 256. As expected, the largest and smallest block sizes are the least efficient.

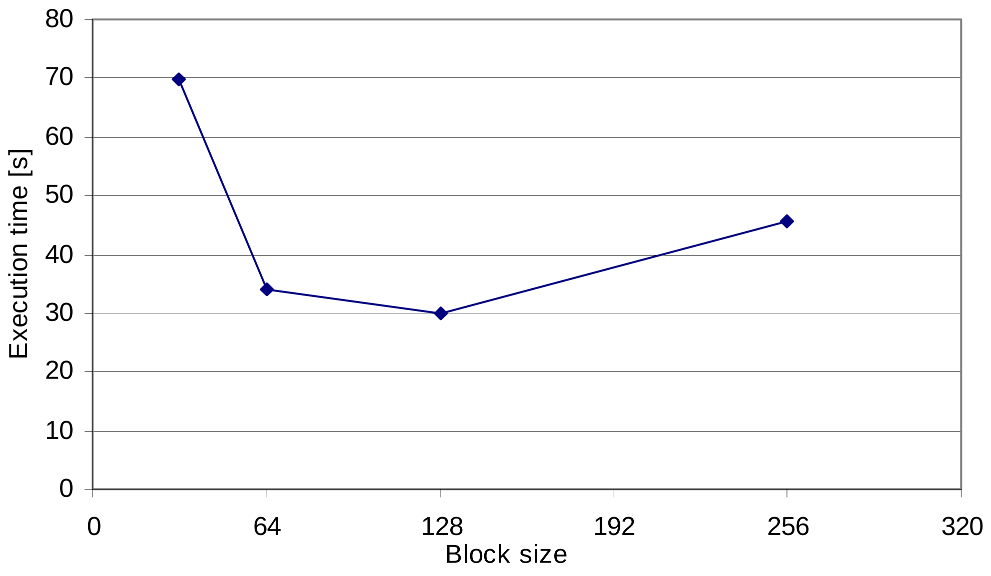


Figure 63. Influence of block size on execution time for the LU decomposition of a 2048x2048 matrix running on 8 Sun Ultra 10 workstations

The implementation of the LU decomposition is pipelined by making use of the stream operation. In order to illustrate the influence of pipelining, we have created a modified version of the stream operations where pipelining is disabled, i.e. the output data objects are not posted before all input data objects have been received. Figure 64 shows the speedup of both the pipelined and non-pipelined variants of the LU decomposition running on the cluster of Sun workstations. The matrix size is 4096x4096, with a block size of 64. The matrix and block sizes allow the same matrix to be used for a number of nodes varying from 2 (32 columns of blocks per node) to 32 (2 columns of blocks per node). The non-pipelined variant very quickly reaches a point where the synchronization imposed by the successive steps of the decomposition limits the overall performance. The pipelined variant however keeps a good efficiency when the number of nodes increases.

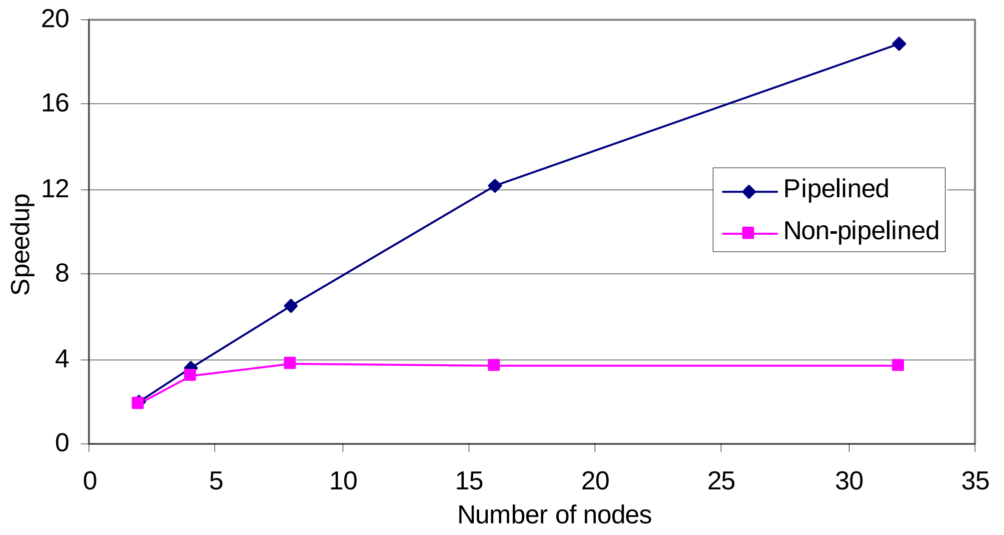


Figure 64. Speedup of the LU decomposition due to pipelining of a 2048x2048 matrix running on Sun Ultra 10 workstations

Figure 65 shows the influence of pipelining on the execution trace of the LU decomposition. When pipelining is enabled, the stream operations can be clearly seen to overlap, and the utilization of the multiplication threads is much higher. This figure was obtained with the DPS trace analyzer tool, which is presented in more detail in Appendix B. A more detailed analysis of the traces is also presented there.

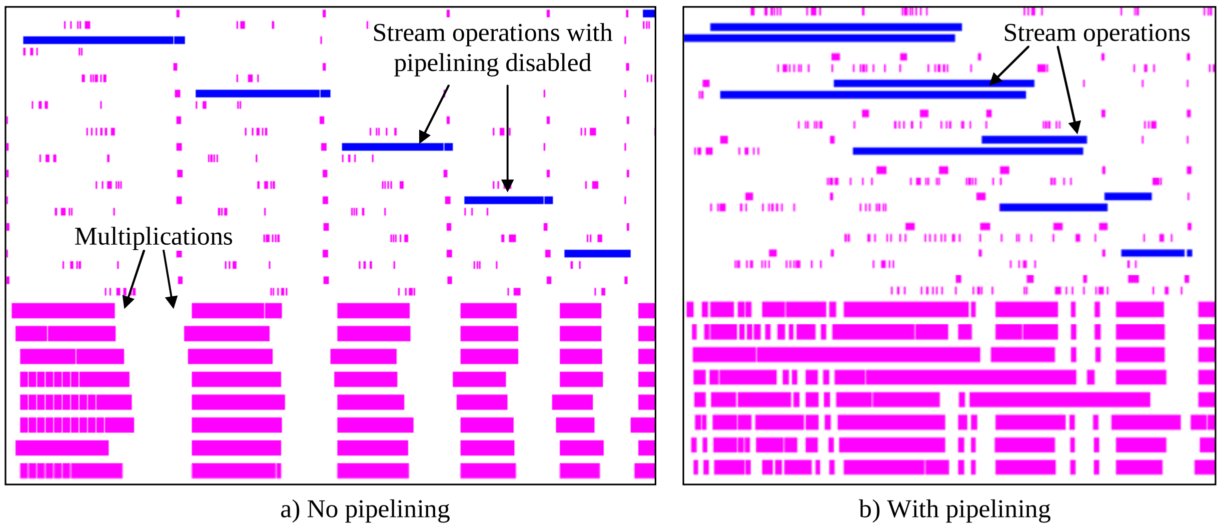


Figure 65. Execution trace of the LU decomposition with a) sequence of split-merge operations and b) overlapping stream operations

The LU decomposition uses flow control for splitting out the multiplications in order to avoid network and memory usage peaks. When flow control is not used, the multiplication requests are all sent out at the same time since the preceding *trsm* computations for solving the triangular system (step 2) complete nearly simultaneously. Flow control prevents these peaks and ensures a more homogeneous filling of the pipeline. Figure 66 shows the difference in application performance when flow control is disabled. All the previous measurements were performed with flow control enabled.

	Execution time [s]		Improvement
	Without flow control	With flow control	
2048x2048, block size 128, 8 nodes	34.84	28.13	19.3 %
2048x2048, block size 64, 16 nodes	27.93	21.11	24.4 %

Figure 66. Influence of flow control on the LU decomposition execution time

4.4.5 Improving LU decomposition performance

The previously presented LU decomposition was based on a direct transposition of the LU decomposition computation steps into a DPS flow graph. The performance of the application can be dramatically improved by modifying two aspects: the flow graph can be redesigned to make better use of the available resources, and the linear algebra routines can be replaced by highly optimized implementations such as those found in ATLAS [Whaley01].

The use of optimized linear algebra routines makes it more difficult to overlap the computation of the matrix block multiplications with the communications required to distribute these multiplications on all nodes, since for reasonable block sizes the communication time to computation time ratio becomes unfavorable, i.e. communication time becomes larger than computation time. In order to remain efficient, the multiplications need to be performed in place on the node where the result will be stored. The multiplication operations and the corresponding thread collection can therefore be removed from the flow graph. Some other operations, such as performing the row flipping on previous columns of blocks after computing the local decomposition of a block of columns, are actually not needed for using the result of the LU decomposition, e.g. for solving a linear system. It is sufficient to take into account that the blocks of the lower triangular matrix have not been updated as a consequence of pivoting when using these blocks in computations. These operations may therefore be removed from the flow graph.

Figure 67 shows the original flow graph implementation of the LU decomposition, and a simplified form that is obtained after applying the above mentioned transformations.

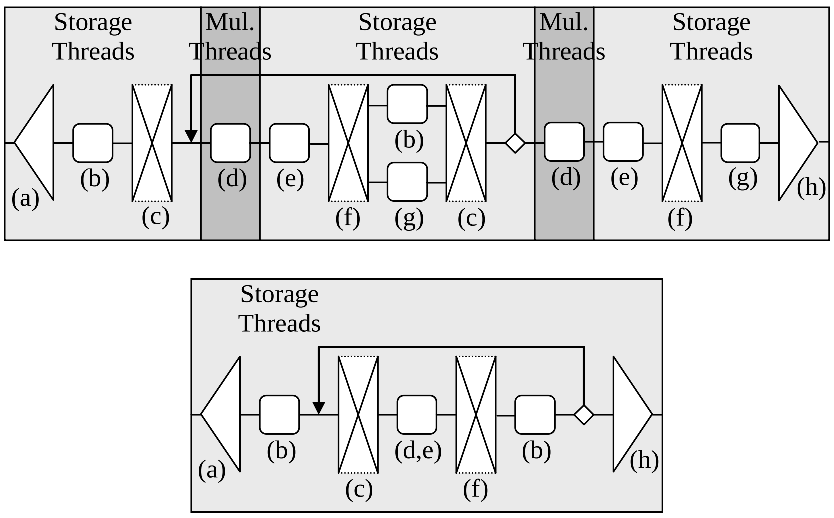


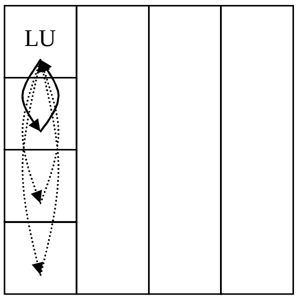
Figure 67. Flow graphs of the LU decomposition on a column-cyclic distribution: initial flow graph (top) and corresponding simplified flow graph (bottom).

The optimized flow graph suffers from load imbalance due to the column-cyclic data distribution, limiting the scalability of the application. In order to ensure that the application scales properly, it is necessary to use a block-cyclic data distribution. When using a block-cyclic distribution, tasks that were performed on a single node on the column-cyclic distribution need to be performed on a distributed data structure. These tasks include the pivot selection for the LU decomposition of a block of columns and the pivoting update performed during the triangular system solves. Matrix blocks also need to be distributed differently in order to perform the multiplications. The following steps need to be performed in order to compute the LU decomposition on a block-cyclic distribution.

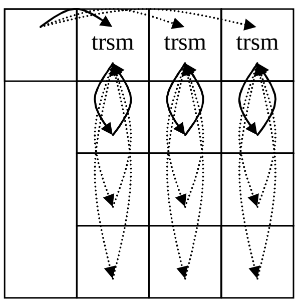
Initial distribution. For the following description, we use an example matrix split into 4 by 4 blocks, with a 2 by 2 block cyclic distribution. In the matrix illustrations, we use arrows to show the required communication patterns. Since multiple blocks reside on a single node, all communications related to blocks residing on the same node can be collapsed into a single communication. The effective communications are shown with solid arrows, whereas the collapsed communications are shown with dotted arrows.

0	1	0	1
2	3	2	3
0	1	0	1
2	3	2	3

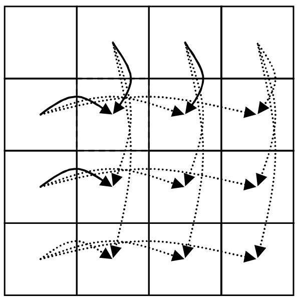
Step 1. In order to perform the LU decomposition of the first block of columns, a pivot row needs to be selected for each column. In order to select the pivot row, all blocks within the column are queried for the best available candidate, which is sent to the uppermost block. The pivot row exchange is performed in the uppermost blocks, and the replacement row is sent back to the appropriate block together with the next query. This process ensures that communications are minimized during the LU decomposition process. The update of the trailing submatrix is performed in the same operation as the pivot selection.



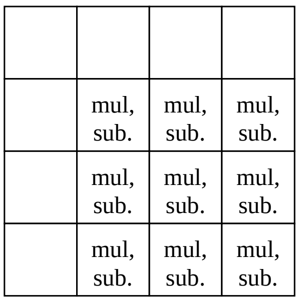
Step 2-a. The result of the LU decomposition and the corresponding permutation matrix is distributed to all other columns of blocks. The permutations induced by pivoting are performed one each column of blocks. Finally, the triangular system $A_{12} = L_{11} \cdot U_{12}$ is solved in parallel in order to obtain U_{12}.



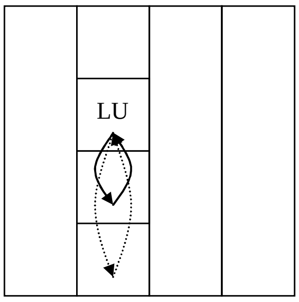
Step 2-b. In order to prepare for the matrix multiplications, the matrix blocks resulting from the LU decomposition of the first block of columns and from the triangular system solve need to be distributed to the nodes where they are required.



Step 3. The multiplications are performed in place on the nodes where the result needs to be subtracted.



Step 1 (next iteration). As soon as the results of all multiplications required for the next column of computations have arrived, the process can restart with the column block LU decomposition. There is no need to wait for the completion of the multiplications for the following column blocks, since there is no dependency before step 2.



The two flow graphs created for performing the block based operations of step 1 and step 2-a are illustrated in Figure 68.

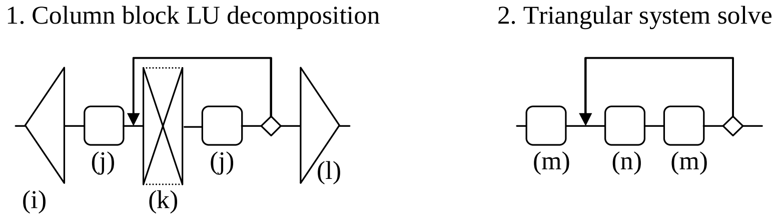


Figure 68. Flow graphs for block LU decomposition and triangular system solve on a block-cyclic distribution. (i,k,l) Perform one step of Gaussian elimination, (j) Select next pivot row, update trailing submatrix, (m) Perform local triangular system solve, (n) perform pivot row exchange with a non-locally stored block

The column block LU decomposition flow graph integrates a loop in order to select the pivot row for each column of the block. The triangular system solve flow graph integrates a loop in order to construct a pipeline for fetching the pivoted rows. Within both flow graphs, the number of required communication steps has been minimized in order to optimize performance. The sequence of computations in the LU decomposition has also been arranged in order to ensure that the computation of the trailing submatrix update effectively overlaps with the transfer of the pivot rows. The complete flow graph for the block-cyclic LU decomposition is illustrated in Figure 69.

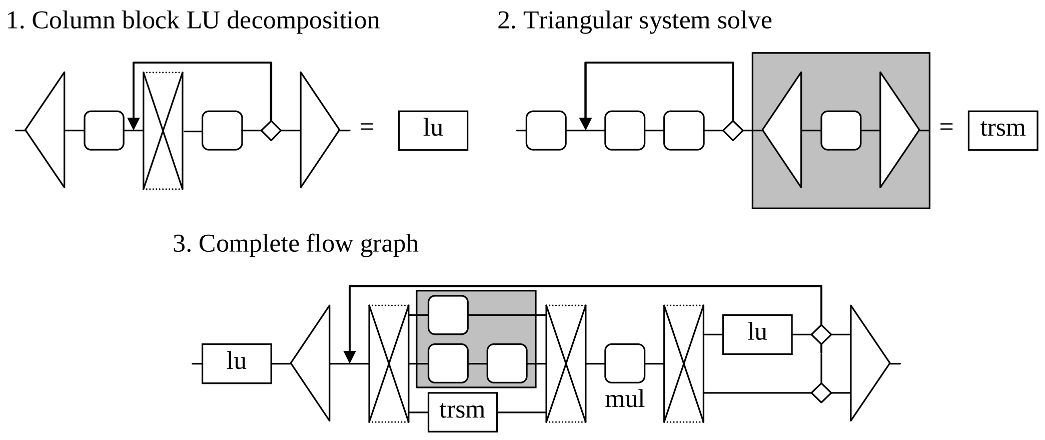


Figure 69. Flow graphs for the complete block-cyclic LU decomposition. The operations in the gray boxes perform matrix block distribution in order to ensure that the required blocks for performing the multiplications are available (step 2-b). The stream operations do not perform any computations, only sequencing.

The LU decomposition with the block-cyclic distribution provides better scalability and performance than the previous column-cyclic distribution. When running on 8 Sun Ultra10 nodes, the block-cyclic LU reaches 1.25 GFlops, whereas the basic column-cyclic approach reached 0.19 GFlops. The improvement is due to the optimized ATLAS linear algebra routines and the higher system utilization made possible by the block-cyclic distribution. The block-cyclic LU decomposition also scales well, as shown in Figure 70.

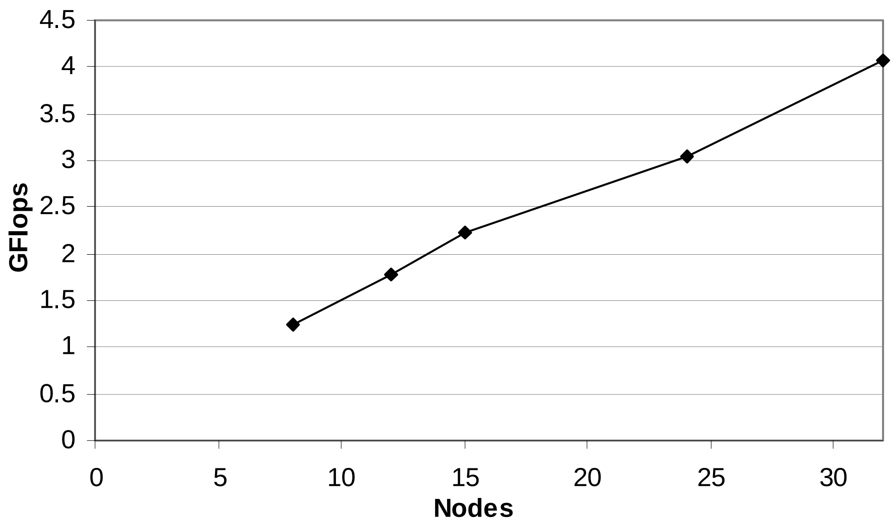


Figure 70. Performance scaling of the LU decomposition with block-cyclic distribution running on the Sun Ultra10 cluster

Finally, we have compared the DPS performance of the LU decomposition relying on the block-cyclic distribution with HPL [Petitet04], an existing LU decomposition implemented using MPI. The results of the comparison are shown in Figure 71.

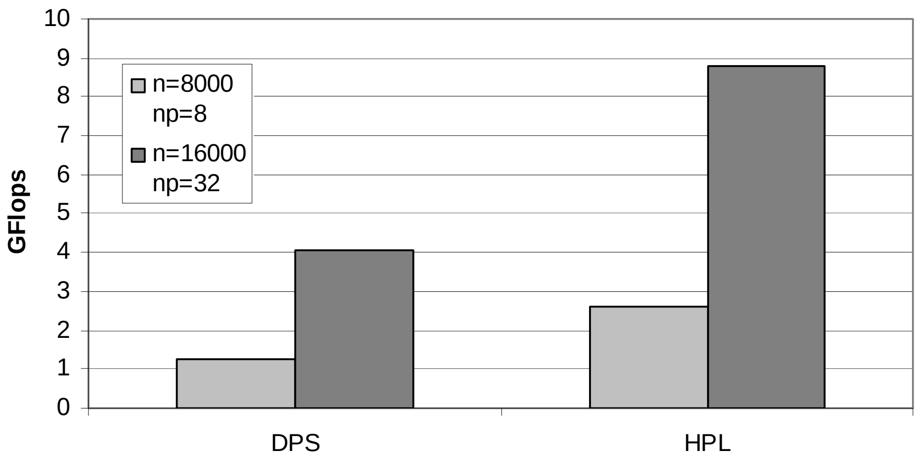


Figure 71. Performance comparison of the DPS LU decomposition and HPL running on 8 and 32 nodes of the Sun Ultra10 cluster

While the DPS LU decomposition reaches only 50% of the performance of the HPL implementation when running on the same machines, it exhibits the same scaling behavior as the number of nodes increases. Both implementations provide a performance increase of a factor of 3.3 when moving from 8 to 32 nodes. The lower absolute performance of the DPS implementation can be explained by several factors. Although both implementations use ATLAS for their linear algebra computations, several parts of the algorithm are not implemented using this library, in particular the column block LU decomposition due to the fact that it is distributed across several nodes. Since this decomposition lies within the critical path of the application, omitting these optimizations has a large influence on the final performance. In addition, DPS incurs a higher internal overhead than MPI due to the need to maintain the flow graph state and the creation and destruction of operations, data objects, and other internal structures. These overheads become visible in latency-critical sequences of operations, in particular during the pivoting phase of the column block LU decomposition. A more detailed analysis of the internal overheads of DPS is presented in Chapter 5.

4.5 Conclusion

The applications presented in this chapter illustrate how the DPS framework can be used to develop complex parallel applications. Flow graphs provide a flexible mechanism for describing applications and reasoning about parallel program structures. The flow graphs can easily be modified by changing the order of operations, by adding or removing synchronization elements, by altering the parallel branches in the graph, or by adding pipelining. Most of these modifications to the flow graph do not require major changes in the source code; it is usually possible to use the same operations within different parallel structures, thereby simplifying experimentation with various parallel program structures.

The examples presented in this chapter also illustrate how the flow graph construction affects the performance of the applications. Features of the DPS framework such as automatic overlapping of communications and computations, flow control and stream operations enable the creation of efficient and scalable parallel applications.

Chapter 5

Inside Dynamic Parallel Schedules

The previous chapters presented the external aspects of DPS, as seen by users of the library writing parallel applications. How DPS works and what exactly happens within a thread or how an operation is actually executed is hidden from the application programmer. Within this chapter, we focus on the internal mechanisms and protocols used within the DPS library, shedding light on what is going on during program execution. We specifically highlight the mechanisms related to data transfer (serialization / deserialization) and the mechanisms for program execution and threading.

5.1 Introduction

The previous chapters focused on the high-level aspects of the Dynamic Parallel Schedules library and parallel program execution. In this chapter, we describe what happens exactly during the compilation and execution of a DPS application.

The C++ language used for developing DPS and DPS applications lacks some essential features that are necessary for the desired behavior of DPS. These features have been implemented within the library by using a combination of template metaprogramming and macros. This chapter explains the concepts and the code behind these templates and macros.

This chapter is split into several parts, highlighting specific aspects of the DPS framework. The first part deals with the construction of flow graphs and thread collections, and the compile-time and runtime validation of the flow graph components. The second part deals with the execution model of parallel schedules, illustrating how DPS handles the execution of operations. The third part examines the implications of supporting malleable applications where threads can be moved from one processing node to another. The fourth part presents the serialization mechanism that is used to transfer data over the network. The fifth part presents additional runtime mechanisms such as object lifetime management and remote procedure calls. The final part presents a performance analysis of the low-level functionality of the DPS framework.

Some aspects that are presented are very technical and require advanced knowledge of the C++ language for a complete understanding. Therefore, the presentations in this chapter are limited to those parts that can be handled with a basic knowledge of C++. The more advanced topics can be found in Appendix C.

5.1.1 Asynchronous execution

An important aspect of the DPS framework is the desire to achieve fully asynchronous execution of the various parts of the framework, in order to maximize the utilization of the available system resources. While this obviously applies to the operations running in parallel in the application's threads, the framework itself should also execute most of its internal tasks asynchronously. This design decision implies that the framework itself is multithreaded, and uses fine-grained locking on all of its internal structures. Almost all subsystems of DPS have their own threads, and communicate using request queues. The request queues ensure that all inactive threads are suspended on synchronization primitives, thereby minimizing their impact on thread switching. None of the threads used by DPS perform active polling on resources.

Using large numbers of threads puts a heavy responsibility on the operating system, which needs to schedule all these tasks, and also track the synchronization primitives. In order to reduce the overall number of threads, only common tasks have their own threads, and less common tasks are executed on threads borrowed from a thread pool. In order to ensure that the whole system remains efficient, it needs to interact with the operating system scheduler, as shown in section 5.7.4.

5.1.2 Object model overview

Since DPS is a C++ library, it is mainly composed of classes that are specialized for particular tasks. The DPS classes can roughly be split into two categories: *user classes* and *internal classes*. The user classes are those that are used by the developer writing an application using DPS. The internal classes are used for the management of DPS, and are not immediately visible to application developers. The internal classes are of interest to anyone wishing to extend DPS, or wishing to use parts of DPS in other projects.

The latest revision of DPS has been designed to be very modular, in order to simplify the development of extensions and the maintenance. Great care has also been taken to eliminate all global variables, singletons, and other constructs living within the global scope. The absence of global constructs makes it possible to run multiple DPS controllers within a single application instance, thereby allowing a complete parallel application to run with full networking within the scope of a single process. Running applications in such an environment is of particular interest for debugging and simulation purposes. Conventional single-process debuggers can then be used for debugging parallel applications running within a single process on a single node, but with all communications features used as if the application were running on a network of nodes.

5.2 Defining parallel schedules

The mechanism for constructing parallel schedules from the developer's perspective has been presented in Chapter 3. The involved objects and their relationships from the developer's perspective are summarized in Figure 72. For the sake of clarity, the shown model is incomplete since specific flow graph constructs such as loops or imported flow graph sections are not mentioned.

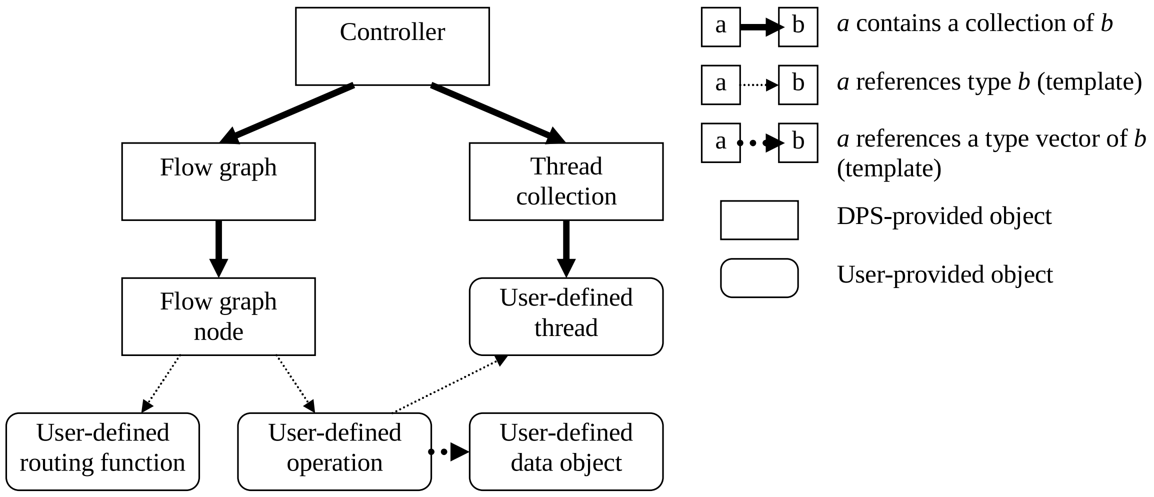


Figure 72. Simplified DPS user object model

The DPS library cannot directly use the representation shown in Figure 72 internally, since templates in C++ are very static. For example, when using parallel components running independently of the client application (section 3.4), a partial flow graph is transferred to the client DPS application, but not all elements that are used in that partial flow graph are necessarily defined within the client application (in particular the operations). Therefore the internal representation in the DPS library uses strings to identify the involved classes, and a class factory that can instantiate objects by name. This class factory is required for many tasks in DPS, in particular within the networking layer, and is described in section 5.2.2.

5.2.1 Data replication

Thread collections and flow graphs need to be known on all the nodes participating in the execution of a parallel schedule. The view of these data structures also needs to be consistent on all the nodes. In order to share these data structures, DPS provides a *replicated data handler* that handles the creation, destruction, and update of these replicated objects [Saito05].

Replicated objects managed by the replicated data handler have the following attributes:

- An internal state - the data inside the object
- An ordered list of owners - the set of nodes on which the object is replicated
- A reference count - used for managing the object's life time

Whenever one of these attributes is updated, all the instances of the object are notified of the update. In order to ensure an absolute ordering of all operations performed on the object, these updates are sequenced by using a master node, which is responsible for sending the update notifications to all owners of the object. The master node for an object is simply the first node within the object's owner list.

The replicated data handler supports the following operations on objects: add owner, remove owner, increment reference count, decrement reference count, user-defined update, and destroy object. Figure 73 shows the sequence of operations performed by the replicated data handler when a new owner is added to the replicated object. The master node keeps a queue of pending requests for an object, and will execute only one request at a time, ensuring that the replicated object stays consistent.

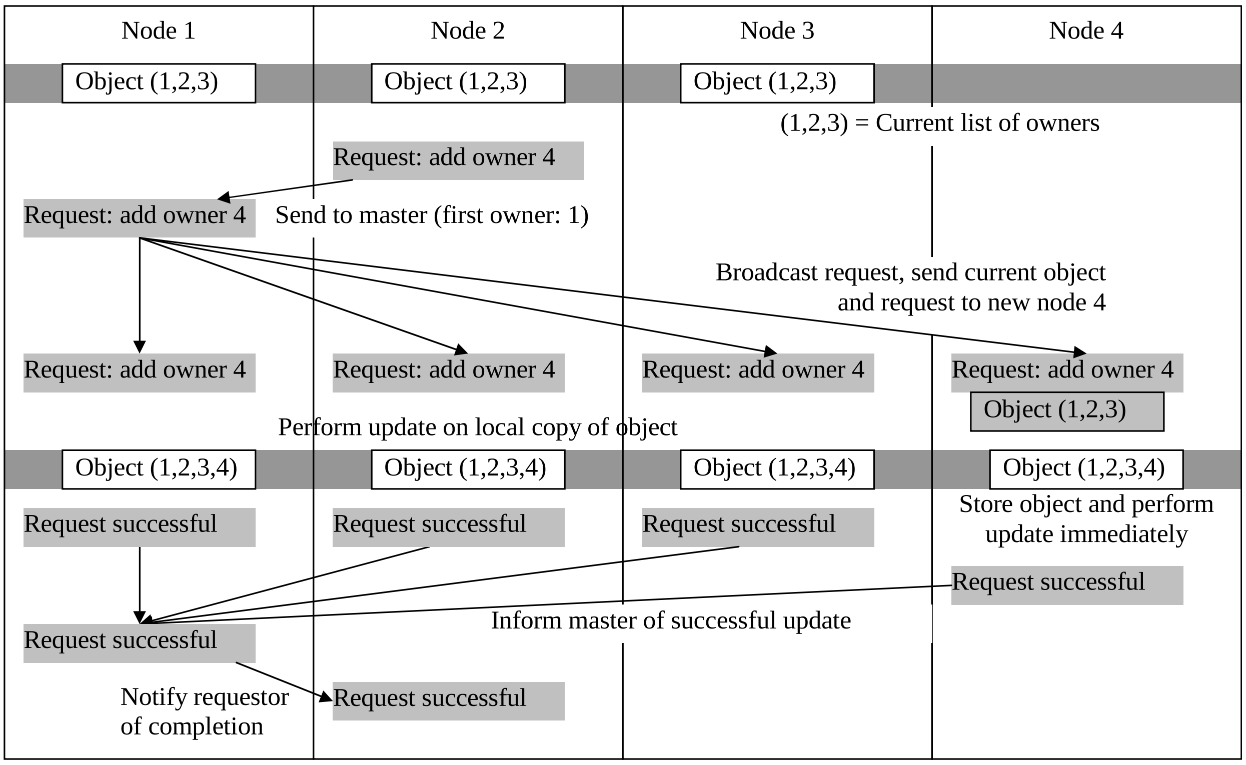


Figure 73. Timeline for the add owner operation on replicated data

Since each node updates its local copy of the object immediately after receiving the update request, the objects may be inconsistent for a very short span of time. This inconsistency is however limited to the changes caused by a single update request, since all updates are sequenced by the master. The DPS components that rely on the replicated data objects have simple mechanisms in place to handle these inconsistencies if they arise. These mechanisms are detailed in Sections 5.2.3 and 5.4.

The other object operations are performed similarly, and are therefore not presented in detail. There are special cases that need to be taken into account. Removing an owner is straightforward, unless the owner to be removed happens to be the master node for the object. In this case, the master will broadcast the request for removal, and deletes its local copy of the object. Any outstanding requests that were in the object's queue will subsequently fail, since the node is no longer the master for that object. The nodes that sent the requests will have to send them again to the new object master on reception of the failure notification.

The reference counting operations can be optimized, since the only condition we are interested in is to find out whether the reference counter is zero or not. Therefore each owner node of the object keeps its own reference count, and only sends a reference count decrement request to the master when its local reference count reaches zero. When the object's master receives such a request, it queries all of the object's owner's reference counts. When the master detects that all owners have a zero reference count, the replicated object is subsequently destroyed on all nodes by a mechanism similar to the one shown in Figure 73.

The replicated data handler also plays an important role in the fault-tolerance mechanism which is described in Chapter 6.

5.2.2 Object instantiation

Another important aspect when using replicated data structures is the need for the framework to be able to instantiate the involved C++ objects without requiring the intervention of any user-provided source code. This requirement can be split into two parts: firstly the type of the object that needs to be constructed must be transferable from one node to another, and secondly, an object of the appropriate type needs to be instantiated.

The first requirement can easily be solved by using string representations of the type names for transfer over the network. The runtime type information (RTTI) functions provided by the C++ language do not return consistent results on heterogeneous systems: the strings differ among different compiler manufacturers. It is therefore necessary for DPS to provide its own mechanism for obtaining a string representation of a type name. This problem can be solved trivially, by including a static method in all classes which returns its name, as well as an optional virtual function in order to support polymorphism:

```
class MyClass
{
public:
  static const char *getTypeName() { return "MyClass"; }
  // Optional virtual function, only present in objects that may be
  // used by pointer reference
  virtual const char *getTypeNameV() { return getTypeName(); }
};
```

These two functions can easily be packaged into a macro, which is the *IDENTIFY* macro seen in all the classes presented in Chapter 3. While this approach is easy to use on simple classes, it is more difficult to generate string representations of classes provided by system libraries and simple types, because it is not possible to modify these types. The more advanced aspects of type identification are described in Appendix C.

The second problem can be solved by using a class factory [Gamma95]. DPS uses the 'traditional' approach for implementing class factories in C++, where classes are registered by using static variables that cause constructors to be called at program startup. These constructors register the name of the class and a function that can be used to construct an instance of the class within the class factory. The class factory can subsequently be used to instantiate the object:

```
MyClass *c = (MyClass *) dps::ClassFactory::create("MyClass")
```

In order to make the registration of the type within the class factory as unobtrusive as possible, the registration should also be performed within the *IDENTIFY* macro. An indirect method using a static variable within a template class is used. The implementation details are presented in Appendix C.

5.2.3 Thread collections

The general object model of thread collections is illustrated in Figure 74. The developer only sees the templated *ThreadCollection* object, which exposes interfaces to the thread collection to the programmer.

The application can also access the data stored within the threads of the collection from within the operations that are executing on these threads.

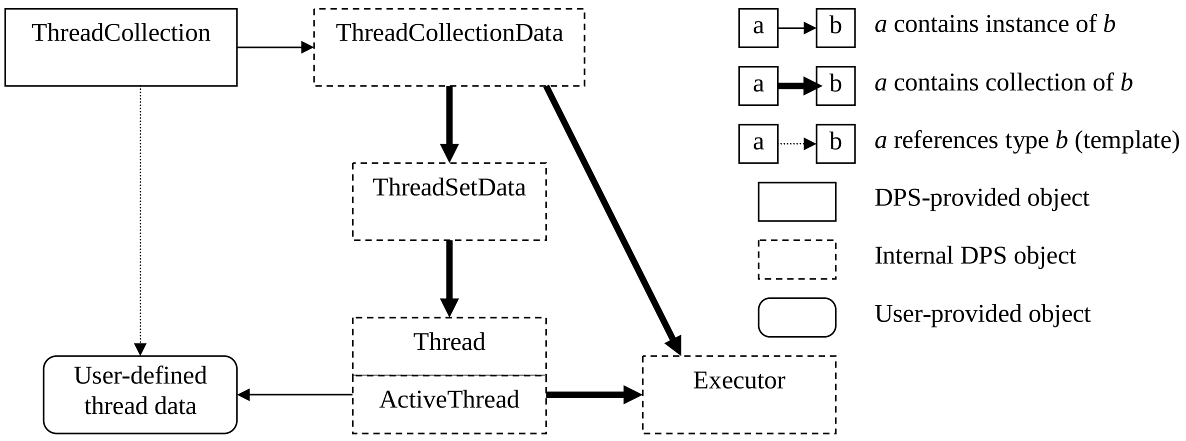


Figure 74. DPS thread collection object model

Internally, the thread collection is represented by the *ThreadCollectionData* object, which contains a collection of *ThreadSetData* objects. The *ThreadCollectionData* object is managed by the replicated data handler, ensuring that it is available on all the nodes that host threads within the collection.

Each instance of *ThreadSetData* represents one thread in the collection. *ThreadSetData* contains one or more instances of *Thread* or *ActiveThread*. The additional level of indirection between the thread collection object and the thread objects is used by the DPS fault tolerance mechanism presented in Chapter 6. With fault-tolerance, each thread in the collection can have a corresponding backup thread, and also backup locations for future use. For the current discussion, we can assume that each *ThreadSetData* contains exactly one *Thread* or *ActiveThread*.

The *Thread* class is a placeholder for DPS threads, and *ActiveThread* derives from *Thread*. The *Thread* class simply contains the address of the node where the thread is located, whereas *ActiveThread* contains the thread's data. A typical layout for a thread collection containing 3 threads distributed on 3 nodes is shown in Figure 75. The addresses contained within the *Thread* instances are used for routing data objects that are posted to the threads.

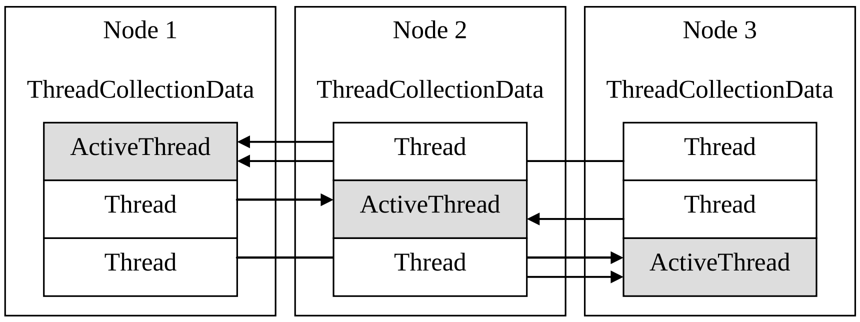


Figure 75. DPS thread collection object model

The type of the thread data that is stored in the *ActiveThread*s is indicated as a template parameter to the thread collection. For thread collections that do not need to store any local thread state data, the *StatelessThread* type can be used. The local thread state data is instantiated within the *ActiveThread* by using the class factory. Beyond the storage of thread data, the thread collections also need to manage the operating system threads that will be used for executing operations. The operating system threads are encapsulated within *Executor* objects. The executors are either owned by the thread collection or by the active threads. When thread aggregation is enabled (i.e. all DPS threads share the same operating system thread(s)), the executors are owned by the thread collection. When thread aggregation is disabled, each active thread has its own executors. The number of executors is also specified as a parameter when creating a thread collection.

```
dps::ThreadCollection<ThreadType> myCollection =
  getController()->createThreadCollection<ThreadType> (
    "collectionName", // name
    2,                // Number of OS threads per DPS thread (default 1)
    false             // Enable aggregation (default true)
  );
```

The operations that the application can perform on the thread collection are reflected on the internal data structures by using the replicated data handler. For example, when a thread is added to the thread collection, the following operations are performed:

1. Add the node on which the thread is to be created to the owners of the thread collection. If the node is a new owner, the thread collection on that node is initialized with a collection of *Thread* instances pointing to the threads that are already in the collection.
2. Send a request to add a new thread. All nodes add a *ThreadSetData* object to the collection. The node on which the thread resides instantiates an *ActiveThread* and the user-defined thread data if applicable. All other nodes create a *Thread* object pointing to the node that is hosting the new thread.
3. The *ActiveThread* creates new executors or inherits those of the thread collection, depending on the aggregation state.

During this update process, some nodes might be aware of the new thread before the new thread has actually been created, since the thread collection update is performed asynchronously. These nodes might post a data object to the new thread, and the node receiving the data object will have no thread to process that data object. The default behavior of DPS in these cases is to return the data object to the sender. The sender simply passes returned data objects through the routing function again, and resends them. By the time the second send is performed, the node hosting the new thread usually has had enough time to create the thread and can correctly process the data object, otherwise this back and forth transfer continues until the thread is created.

Removing a thread involves the removal of the corresponding *ThreadSetData* and matching *Thread*s or *ActiveThread*s on all owners. Again, a potential inconsistency that can arise during the update is that a node sends a data object to a thread that is currently being destroyed. When the data object arrives at the target node, and the thread has already been removed, it is simply returned to the sender, where it is re-routed. In the meantime, the sender should already have been updated with the new composition of the thread collection, and selects a valid thread this time.

5.2.4 Flow graphs

The general object model of flow graphs is illustrated in Figure 76. The developer sees two distinct parts: the flow graph builder and its templated *FlowgraphNode*, *FlowgraphLoop* and *FlowgraphSection* elements, and the *Flowgraph* object that is returned by the controller when the flow graph is created. The template arguments of *FlowgraphNode* and of the referenced operation provide access to all the important type information for the flow graph node.

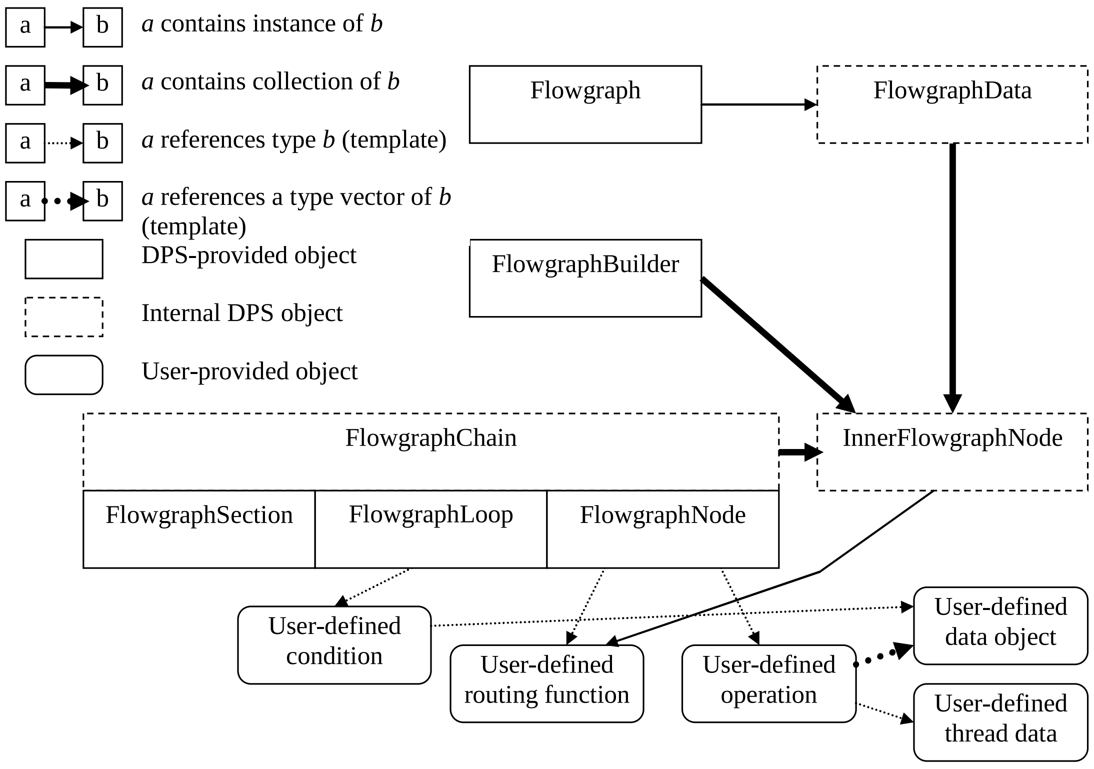


Figure 76. DPS flow graph object model

Internally, the flow graph is represented by the *FlowgraphData* object, which contains a collection of *InnerFlowgraphNode* objects. The *FlowgraphData* object is managed by the replicated data handler, ensuring that it can be made available on all nodes that need it. The *InnerFlowgraphNode* object contains all the type information present in the template arguments of the various flow graph node types in string form, allowing all the referenced objects to be created with the class factory. Of the referenced objects, conditions and routing functions are created immediately, since they do not store any internal state and operate only on the input data objects. The functionality of these objects is therefore thread-safe and can be invoked as required without needing to recreate the objects for every call. The operations however need to be created for each call, since they can contain an internal state and are therefore not thread-safe.

Three types of elements can be inserted into the flow graph builder: operations, loops, and imported flow graph sections. These elements derive from a common base class, *FlowgraphChain*. The *FlowgraphChain* template parameters specify the allowed input and output data object types for the element it represents. For loops and operations, the data object types are retrieved from the loop and operation classes, whereas they have to be specified explicitly for imported flow graph sections. The type matching is performed when the elements are combined with the overloaded right shift (>>) operator. The shift operator also creates a list of the *InnerFlowgraphNode* elements as they are connected together. This list of internal flow graph nodes is subsequently built into a graph when it is added into the *FlowgraphBuilder* with the overloaded = and += operators.

Once a flow graph has been constructed, it can no longer be modified. If a different flow graph is desired within a running application, the application is expected to create a new flow graph. Therefore the only role of the replicated data handler with respect to flow graphs is to ensure that they are available on all the nodes that need them. For this simple replication where no updates are performed, no temporary inconsistencies can occur.

5.2.5 Type matching

The template arguments to the *FlowgraphChain* class do not necessarily represent single types, since split operations and stream operations can produce output data objects of different types in order to introduce branches into the flow graph. The data object types are therefore specified by using type lists

[Alexandrescu01]. Type lists are represented by the templated class *tv*, which can take between one and 5 arguments as follows.

```
dps::tv<MyType>                    // Type vector with one element
dps::tv<MyType1, MyType2, MyType3> // Type vector with three elements
```

The DPS library provides metaprogrammed templates for performing type matching between individual types, finding types in type vectors, and finding a common type between two type vectors. All these template structures contain a member *result* that has a non-zero value if a match is found.

```
// Simple type matching
template<typename t1, typename t2> struct MatchTypeType;

// Find type in type vector
template<typename tv, typename t> struct MatchTypeTypeVector;

// Find common type in two type vectors
template<typename tv1, typename tv2> struct MatchTypeVectorTypeVector;
```

In the implementation of the right shift operator used for combining *FlowgraphChain* items, the *MatchTypeVectorTypeVector* template is used to check for a common token type between the output types of the first chain and the input types of the second chain. If no match is found, the constant evaluates to a division by zero, thereby generating a compiler error. The following source code is the complete implementation of this test:

```
// Connect two flow graph chains with the >> operator
template<typename i1, typename o1, typename i2, typename o2>
  FlowgraphChain<i1,o2> operator>> (const FlowgraphChain<i1,o1>& chain1,
                                    const FlowgraphChain<i2,o2>& chain2)
{
  // Check for a common token type between the output types of the
  // first chain and the input types of the second chain.
  // Generate a divide by zero compile time error if no match is found.
  enum { CheckForCompatibleTokenTypesBetweenOperations =
           1 / MatchTypeVectorTypeVector<o1,i2>::result
  };

  // Construct new chain by concatenating chain1 and chain2
  return FlowgraphChain<i1,o2>(chain1,chain2);
}
```

The internal workings of the type lists and the type vector comparison templates are presented in Appendix C.

5.3 Parallel schedule execution

The previous section focused on the internals of the declaration of flow graphs and thread collections. These declarations provide the DPS library with the necessary data structures and threads for executing parallel schedules. The replicated data handler ensures that these structures can be made available on all nodes that require them, and the generic string-based descriptions contained within these structures ensure that they can be successfully constructed anywhere.

Execution of a parallel schedule starts by posting a data object to a flow graph, thereby causing the invocation of the flow graph's first operation. This operation will subsequently post more data objects, which will cause the invocation of further operations, and so on until the execution of the parallel schedule is terminated. The DPS library ensures that the operations are executed within the desired context. Since the execution model is fully driven by the data objects, it makes sense to use these data objects as the primary vector for state information of the running parallel schedule. The state information needs to convey all the information required in order to successfully execute the appropriate operation that will process the data object within the correct thread context. The required state information is added to the data objects by packaging these data objects inside *tokens*. The tokens are self-contained data structures, allowing DPS to construct the execution environment required for processing the contained data object within its parallel schedule. The state information stored within tokens is composed of the following elements:

- *A unique flow graph identifier* – This identifier enables the node that is currently in possession of the data object to obtain a copy of the replicated flow graph data object for the parallel schedule that the data object is part of.
- *A flow graph node identifier* – This identifier uniquely identifies the node within the flow graph that indicates which operation is to be performed on the data object. This operation can be instantiated by using the class factory, or an existing instance of the operation can be reused. The flow graph node also indicates which thread collection contains the thread within which the operation is executed.
- *A thread identifier* – This identifier is used to select the appropriate thread within the thread collection on which the data object is to be processed.
- *A token identifier* – This identifier is a hierarchical history of the operations the data object has traversed, with an identifier of the parallel schedule at the root of the hierarchy. The token identifier is used to keep together related data objects, ensuring that the data objects created from the data objects posted by a split operation are processed within the context of the same merge operation.

This state information is sufficient to create the correct processing environment for a given data object and recover the required flow graphs and thread collections. The fields within the token are filled out by the operation that posted the data object. The first three fields can be filled out by examining the flow graph, whereas the token identifier is constructed hierarchically. The next sections explore how this information is initialized and used.

5.3.1 Flow graph and thread identifiers in the token

The first three fields of the token are used by DPS to send the data object to the appropriate thread for processing, and to determine which operation should be executed. These fields are filled out as the data object traverses DPS' data processing pipeline. The procedure for filling out the fields is the following:

1. Select the flow graph (*flow graph identifier*)
   a. If the data object is passed to *callSchedule* (i.e. a new parallel schedule is being invoked), the flow graph is available as a function argument (see section 3.5.11).
   b. If the data object is posted by an operation, the flow graph is the same as the one specified in the data object that caused the operation to execute, it is therefore copied over from the token of the operation's input data object.
2. Find the successor node within the flow graph (*flow graph node identifier*)
   a. If the data object is passed to *callSchedule*, the successor is the head node of the flow graph passed as argument.
   b. If the data object is posted by an operation, the successor is the next node on the flow graph where the corresponding operation can take a data object of this type as input. If the flow graph node has no successors, the end of the parallel schedule has been reached, and a special flag is set to indicate that the data object should be returned to the caller of the parallel schedule.
3. The thread identifier is determined by calling the routing function associated with the successor flow graph node. The flow graph node also specifies a thread collection within which the thread is selected. The routing function returns the index of the desired thread in the collection, possibly by making use of the fields present in the data object and the size of the target thread collection.

The thread index returned by the routing function is converted to the target thread identifier within the thread collection.

Once the target thread identifier within the target thread collection is known, the node on which this thread resides can be looked up in the thread collection data structures. If this target node is not the local node, the token is sent asynchronously over the network to the target node. Once the tokens are on the node where their target thread resides, they are stored in a queue within the executor of their target thread.

5.3.2 Token identifiers

DPS uses token identifiers in order to ensure that merge operations receive related data objects. Token identifiers are basically a simple list of integers. For leaf operations, the output token identifier is identical to the input token identifier. For split operations, the output token identifier contains all the elements of the input token identifier, and an additional integer indicating the number of data objects that have already been posted by this split. The additional integer ensures that all data objects posted by a given split operation are unique, yet have a common part, since everything except the last number is identical. For merge operations, the output token identifier is a copy of the input token identifier without the last element. Therefore the output token identifier of a merge operation is identical to the input token identifier of the corresponding split operation. The token identifiers used in the border exchange process of the previously illustrated game of life are shown in Figure 77.

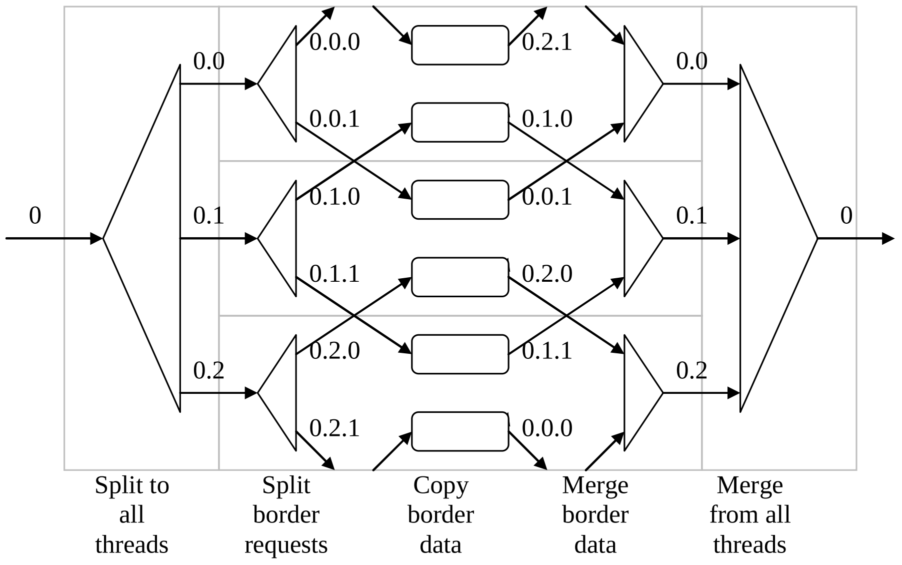


Figure 77. Token identifiers in the border exchange process of the game of life.

The initial value stored in the token identifier is actually the identifier of the parallel schedule. This allows the same flow graph to be used multiple times simultaneously within separate parallel schedules, without having any collisions between identifiers. The parallel schedule identifier is generated by using a counter that is incremented for every invocation of a parallel schedule using a given flow graph.

Stream operations behave like a combination of a split and merge operation. They therefore have the same token identifier length on their input and output. The last element of the output token identifier follows the counting pattern seen in split operations. The following discussion refers only to split and merge operations; all the presented concepts are also applicable to stream operations, since they combine the functionality of both split operations and merge operations within a single operation.

The token identifier is used to control the instantiation of new operations when data objects need to be processed. Split and leaf operations are always instantiated, since they only process a single input data object. Merge operations however need to process multiple input data objects within the same context. This context is identified by examining the token identifier of incoming data objects to the merge operation. Since split operations add one unique element to the end of the token identifier of all data objects they post, all the data objects that share a common identifier list with the exception of the last element are processed by the same merge operation.

The token identifiers provide sufficient information to identify when new merge operations need to be created, but do not indicate when these merge operations should terminate. The merge operation needs to terminate (i.e. *waitForNextDataObject* returns *NULL*) when it has received a number of data objects equal to the number of data objects posted by the corresponding split operation. This information is transferred by setting up a feedback loop between the merge operation and its corresponding split operation, as illustrated in Figure 78. This mechanism needs the split operation to remain available until the exchange with the merge operation has been performed, even after the split operation has terminated its execution. The appropriate split operation can be found by using the token identifier, since it has the same input token identifier as the merge operation's output data object.

The notification is carried out by using special data objects, *NotifySplit* (sent from the merge operation to the split operation) and *NotifyMerge* (sent from the split operation to the merge operation), that are packaged into tokens like any other data object. These data objects are however processed by the framework on arrival rather than passed on to the user-provided operation.

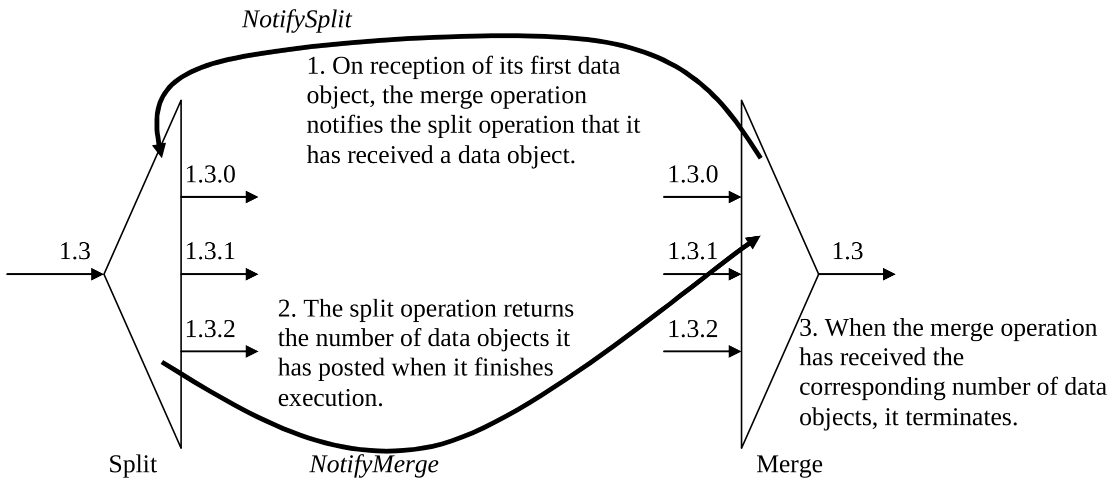

NotifySplit
1. On reception of its first data object, the merge operation notifies the split operation that it has received a data object.
1.3
1.3.0
1.3.1
1.3.2
1.3.0
1.3.1
1.3.2
1.3
2. The split operation returns the number of data objects it has posted when it finishes execution.
3. When the merge operation has received the corresponding number of data objects, it terminates.
Split
NotifyMerge
Merge

Figure 78. Split-merge communication for determining merge operation lifetime.

This feedback mechanism is also used by DPS to detect routing errors within the application. If the routing function leading to a merge operation is erroneous, data objects that should be processed in the same merge might be sent to different nodes. In this case, the merge operation will be instantiated once on each node, and each instance will notify the split operation of its existence. When the split operation receives multiple notifications from different merge operations, the DPS library knows that the application is using an erroneous routing function and exits with a fatal error message.

5.3.3 Token identifiers in complex flow graphs

Using simple hierarchical numeric identifiers is sufficient in simple flow graphs without loops, since the token identifiers are always unique for a given operation in the flow graph. However, when loops are introduced, data objects are passed through the same flow graph parts multiple times. The corresponding invoked operations need to be re-instantiated since within the loop, the operations are replicated. Therefore, an additional distinguishing factor needs to be added to the token identifier. In this case, a counter is used, which counts the number of operations that have been performed on a data object at a given level of the hierarchy. Since the operation counter is incremented at least once for every pass of a loop, it ensures that the resulting token identifiers are unique when the same flow graph node is reused.

Figure 79 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 token identifiers circulating along the graph edges. The stream operation has exactly the same behavior on the counter as a split-merge operation pair.

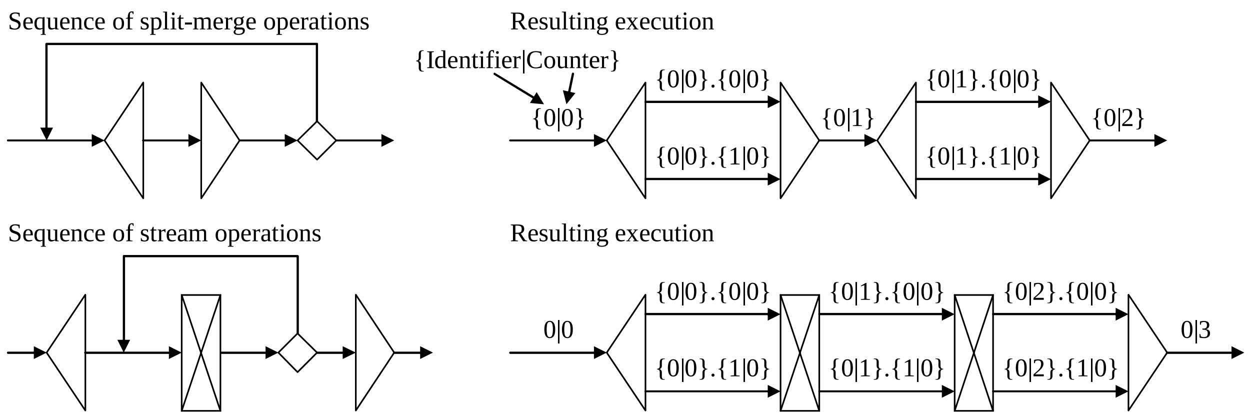


Figure 79. Loops and resulting parallel schedule execution (operations have been omitted for simplicity)

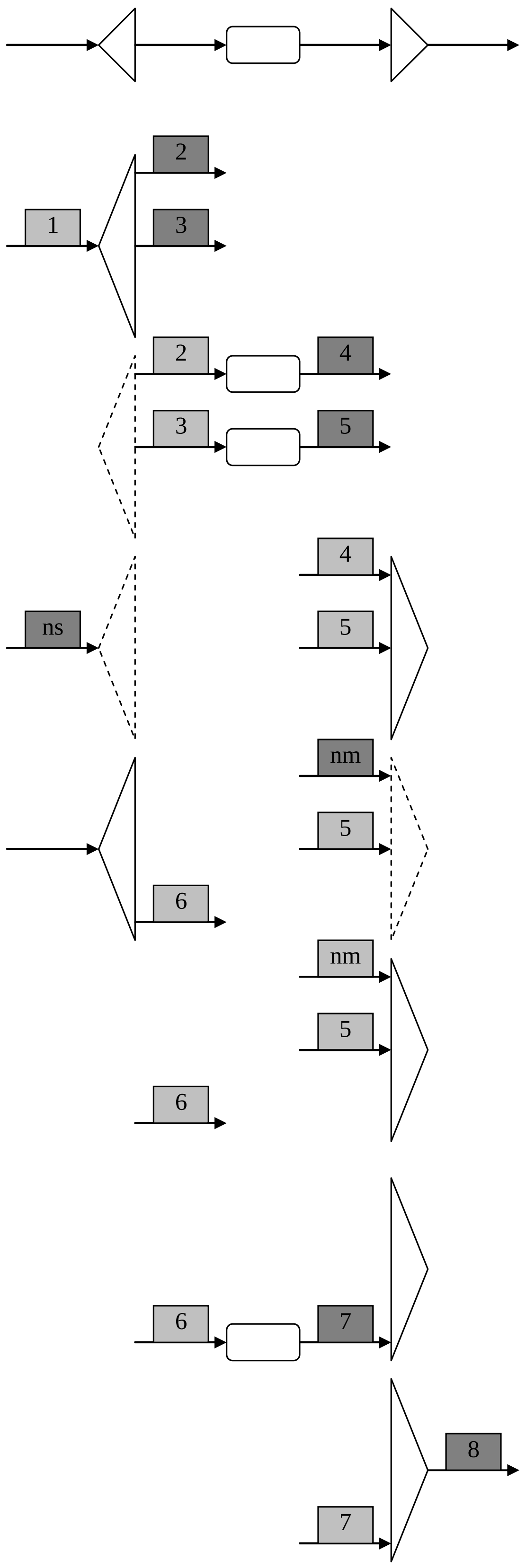
The only loop case where the operation counter is strictly necessary is when the loop contains stream operations. In the split-merge case, the merge operation is a strict global synchronization point where all preceding operations need to have terminated before it can terminate. Therefore the first merge operation can never coexist simultaneously with the second merge operation, since it will terminate on posting the data object needed to instantiate the second split. Since the two merge operations never coexist, it does not matter if the token identifiers within the parallel schedule are repeated, since there can be no ambiguity about the target merge operation. In the case of the stream operations, however, there is no synchronization; therefore all the stream operations along the pipeline can exist simultaneously. Therefore the counter is necessary to ensure that the data objects are routed to the correct stream operation.

Adding the counter to the token identifier ensures that the token identifiers truly become unique within the execution of a parallel schedule, which is a property that is highly desirable for the fault tolerance mechanism presented in Chapter 6.

5.3.4 Flow control and load balancing

Within parallel schedules, the feedback loop used for handling the termination of merge operations also forms the basis for the implementation of the flow control mechanism. When flow control is enabled, the split operation will only post the number of data objects indicated in the flow control parameter before suspending its execution. The corresponding merge operation will start the feedback process when it starts receiving data objects, indicating how many data objects it has already received within the posted *NotifySplit* data object, thereby freeing the split operation to post more data objects. The frequency with which the merge operation sends notifications to the split operation is set by the flow control group size, indicating how many data objects should be received before the next notification is sent. The flow control group size allows limiting the amount of network traffic created by the flow control mechanism.

The load balancing feature is also based on the feedback loop, and relies on the flow control mechanism. Whenever a split operation uses the load balanced routing function, it initially sends out its data objects with a round-robin distribution up to the flow control limit. The value for the target thread indices is stored within the token, and preserved until the corresponding data object reaches the merge operation. The merge operations sends these stored thread indices back to the split operation within the *NotifySplit* data object, thereby allowing the split operation to reuse the indices of threads that have already returned results to the merge operation. This reuse of indices effectively ensures that every thread is always in possession of a constant number of data objects that need to be processed.



Flow graph declaration. All operations are mapped on to the same thread, and therefore run within the same executor. The flow control level on the split operation is set to two. The parallel schedule is invoked with object 1.

1 ← Current queue state in executor

Execution starts by instantiating the split operation, and passing to it the input data object. The split operation posts two output data objects, and is suspended on flow control.

2 | 3

The next data object in the queue is object 2, therefore the executor instantiates the leaf operation and executes it, producing object 4.
The process is repeated with object 3, producing object 5.

4 | 5

The next data object in the queue is object 4, therefore the executor instantiates the merge operation and executes it. The merge operation sends a notification to the split operation (*ns*), informing it that one data object has been consumed. The notification is placed at the head of the queue. The merge operation is suspended on *waitForNextDataObject.*

ns | 5

The executor processes the notification (*ns*) by resuming the split operation. The split operation posts an output data object and terminates. It sends a notification (*nm*) to the merge operation to indicate that it has posted 3 data objects

nm | 5 | 6

The executor processes the notification to the merge operation (*nm*). The merge operation now knows that it will receive 3 data objects, and that it can stop sending flow control notifications to the split operation.
The next data object is 5, which is consumed by the already instantiated merge operation, which again suspends itself on *waitForNextDataObject.*

6

The next data object in the queue is object 6, therefore the executor instantiates the leaf operation and executes it, producing object 7.

7

The merge operation consumes object 7, and on calling *waitForNextDataObject* receives *NULL*, since this was the last data object for this merge. It posts its output data object 8.
The execution of the parallel schedule is complete.

Figure 80. Simple parallel schedule execution with flow control

5.3.5 Execution model

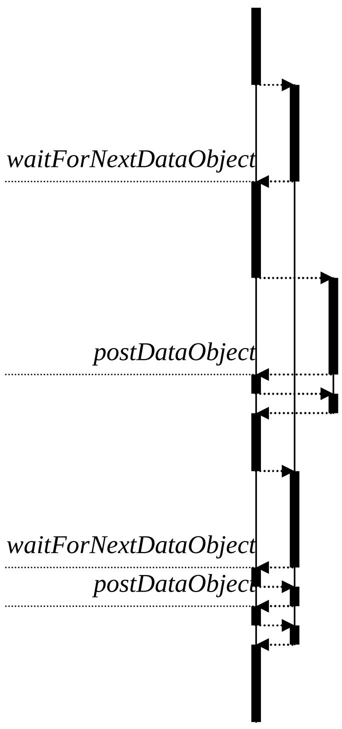
All operations within parallel schedules are executed in the context of the operating system threads owned by the executors. Executors can either be specific to a single DPS thread or shared by multiple DPS threads when the aggregation mechanism is used. Each executor owns one operating system thread and stores a queue of tokens containing the data objects that need to be processed within their context. The executors simply take the tokens out of their queues, examine the token's fields in order to discover which operation needs to be executed on which DPS thread, instantiate the operation as required using the token identifier, and execute the operation on the provided data object. The tokens are processed on a first in, first out basis. Since the executors run on their own operating system threads, they run fully asynchronously, interacting with the rest of the system only through their data object queues.

Let us examine the DPS execution model with a simple example, involving a simple split-operation-merge flow graph. The split operations posts three data objects and has a flow control level of two, which will cause it to be suspended during the execution of the parallel schedule. The complete execution is illustrated in Figure 80. The execution is shown on a fully unfolded view of the flow graph, but the internal representation within DPS consists simply of two lists: the tokens in the executor's queue, and the operations that are currently suspended in the thread.

This example illustrates how DPS handles the notification messages circulating between the split and merge operations: they are also packaged into tokens, and they pass through the same queues as user-defined data objects. The only difference is that these notifications have a higher priority, and are placed at the beginning of the queues rather than at the end.

When the parallel schedule is running on multiple threads with multiple executors, the only difference to the present example resides in the asynchronous processing of all the queues. The actual location of the threads is of no consequence to the execution model, since the tokens are transferred by the runtime system to their target node.

The example in Figure 80 shows that there are many circumstances within DPS where an operation needs to be suspended because it cannot continue execution within the current context. The most common case is the merge operation when it is calling the *waitForNextDataObject* method: the merge is suspended until the next data object is available or the number of data objects posted by the corresponding split



Initial state of executor, running in its own stack frame

Data object arrives; the executor creates the corresponding merge operation with its own stack frame. Execution switches to merge operation stack frame.

The merge operation completes processing of the data object. It calls *waitForNextDataObject*. Execution switches back to the executor stack frame, and the merge operation is suspended.

Another data object arrives; the executor creates the corresponding leaf operation with its own stack frame. Execution switches to the leaf operation stack frame. The leaf operation calls *postDataObject*. Execution switches back to the executor stack frame in order to process the posted data object, and returns immediately to the leaf operation stack frame. The leaf operation terminates, returning control to the executor stack frame. The executor destroys the leaf operation.

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

This was the final data object for the merge operation. The executor returns NULL to *waitForNextDataObject*, the merge operation posts its output data object, passing control back to the executor, who returns it immediately to the merge operation. The merge operation terminates, returning control to the executor stack frame. The executor destroys the merge operation.

Figure 81. Stack frame management in DPS threads

operation has been reached. A further suspension case includes suspending the split operation because it has called *postDataObject* to post another data object and has reached the flow control limit. Since the operations are composed of simple sequential C++ code, DPS needs to keep the complete execution context (the *stack frame*) of the operation when it is suspended so that it can be restarted from the same point in the future [Tanenbaum01]. The functions required for managing stack frames are provided by the underlying operating systems (*fibers* in Windows, *ucontext* in UNIX variants). The DPS library creates one stack frame for each executing operation. When an operation is suspended, it is stored within the DPS thread it is associated with, and DPS switches back to the executor's main stack frame. Figure 81 shows the stack switching in effect for an executor that processes three data objects, 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.

5.3.6 Parallel schedule startup and termination

When a parallel schedule is invoked by calling *callSchedule*, a token is allocated for the initial data object and execution can begin. In addition to the previously shown fields, the token also stores the node address of the calling site of the schedule. This information is replicated from one token to the next as the schedule executes. This allows the last data object that is posted from the last operation in the schedule's flow graph to be routed back to the node where the schedule was called. This last data object is returned by the *callSchedule* function.

5.4 Support for malleability

DPS enables an application to move threads from one node to another, fully asynchronously, without interrupting the program execution. The thread displacement operation is synchronized within a request to the replicated data handler. Figure 82 provides an overview of the thread displacement process.

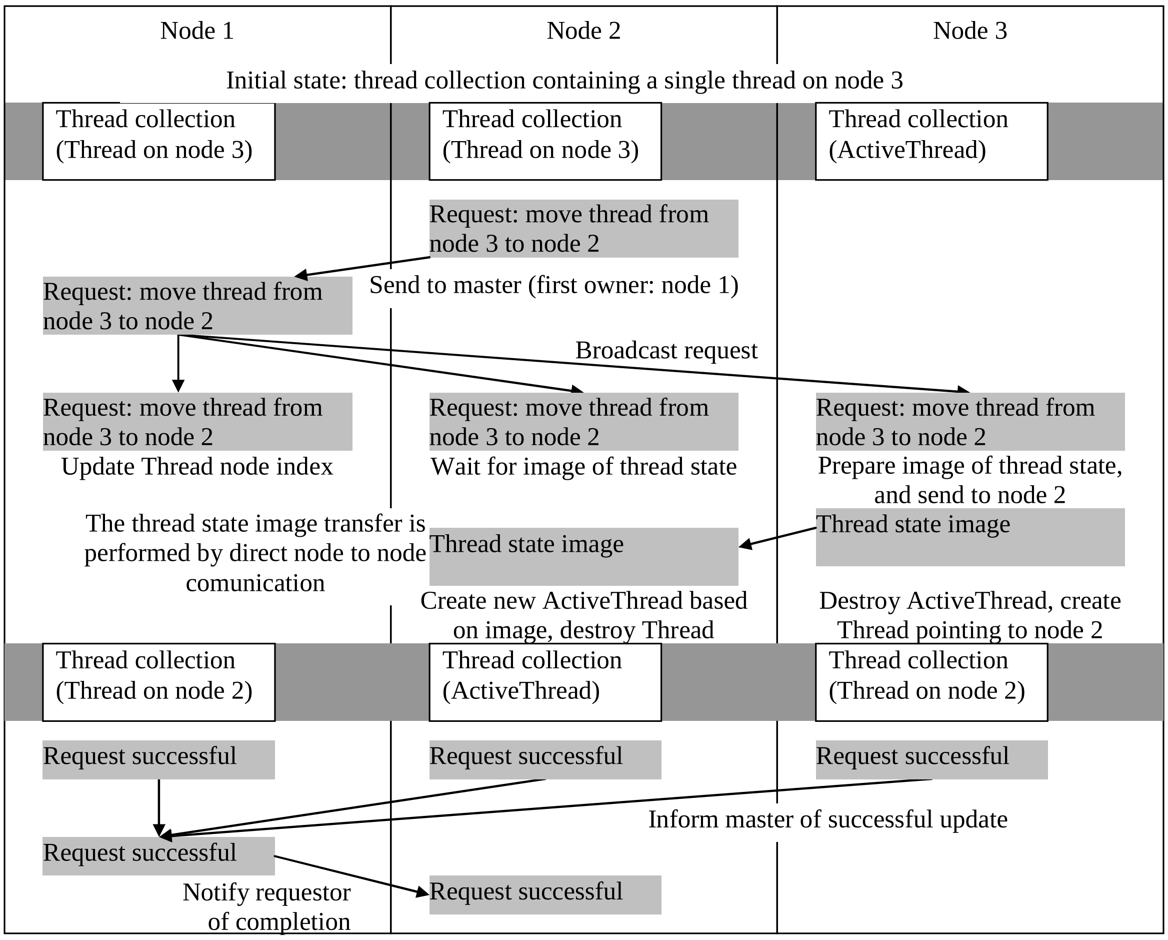

Node 1
Node 2
Node 3
Initial state: thread collection containing a single thread on node 3
Thread collection (Thread on node 3)
Thread collection (Thread on node 3)
Thread collection (ActiveThread)
Request: move thread from node 3 to node 2
Send to master (first owner: node 1)
Request: move thread from node 3 to node 2
Broadcast request
Request: move thread from node 3 to node 2
Update Thread node index
Request: move thread from node 3 to node 2
Wait for image of thread state
Request: move thread from node 3 to node 2
Prepare image of thread state, and send to node 2
The thread state image transfer is performed by direct node to node comunication
Thread state image
Thread state image
Create new ActiveThread based on image, destroy Thread
Destroy ActiveThread, create Thread pointing to node 2
Thread collection (Thread on node 2)
Thread collection (ActiveThread)
Thread collection (Thread on node 2)
Request successful
Request successful
Request successful
Inform master of successful update
Request successful
Notify requestor of completion
Request successful

Figure 82. Using the replicated data handler for thread displacement

The thread state image that is transferred during the thread displacement process needs to be consistent. Therefore, the node that is hosting the active thread freezes all the executors that are being used by the thread to be displaced. When entering the frozen state, the executors continue the execution of the currently running operation until it is either completed or suspended. This ensures that no operations are currently running on this thread. All other executors can continue execution. DPS then collects all the elements that are part of the thread state: the current state of the user-defined thread data structure, all the tokens that are currently in the thread's execution queue, and the current state of all suspended operations (i.e. split operations waiting on flow control or merge operations waiting for their next input data object). All these elements are packaged into a single thread state object that is sent to the new owner of the thread. Once the packaged object is complete, the active thread is destroyed and the executors that were attached to that thread can resume execution if they are also executors of other threads. Otherwise the executors are destroyed. The displaced thread is then recreated on the new node based on the packaged object, and executors are attached to this new thread in order to allow execution to proceed.

Since execution of the remainder of the parallel application is not halted during the thread displacement process, it is possible that operations running on other threads generate data objects that are routed to the thread that is being moved. Since the thread collection is not globally locked during its update, some owners will know the previous state, whereas others will already have the new state. Therefore tokens will arrive both on the previous owner node and on the new owner node. Since the tokens identify the target thread for execution by the thread identifier, these nodes will know where the thread currently is supposed to be located, and forward the token to the correct node on behalf of the sender. This mechanism ensures that data objects never get lost, even when the thread collection is in an inconsistent state.

During thread displacement operations, potential inconsistencies in the replicated data object could arise. For example, during the transfer of the *ActiveThread* from one node to another, the thread will not have any active site, since the previous site contains a new *Thread* that is already pointing to the new location, whereas the new site still contains the old *Thread* pointing to the previous location. In this case, a data object being sent to the thread from the new location will initially be sent to the old location. Once it arrives at the old location, it will be posted to the *Thread* there, which will forward it back to the new location. If the transfer of the *ActiveThread* is complete by then, the data object will then be correctly processed by the thread in its new location. This simple ping-pong mechanism can efficiently handle any inconsistencies, ensuring that data objects always end up at the right location. The additional communications overheads are not significant, since moving threads is not a frequent operation. This recovery from inconsistencies is very similar to the general mechanism used when adding or removing threads from a thread collection.

5.4.1 Requirements for malleability support

In order to support malleability, the DPS application needs to respect a few additional requirements: the thread data structures need to be serializable, and the operations that can be suspended (i.e. split, merge and stream operations) need to be restartable. The serializability of thread data structures is achieved by using the same techniques as for data objects, i.e. providing specialized serialization code or using the library's *dps::AutoSerial* object (discussed in section 5.5).

Operations that need to be resumed upon displacement of their thread need to be able to serialize their internal state as well. This is usually achieved by removing local variables used as loop counters, etc. and storing them in the operation class instead. When the operation is restarted, it will not be passed an input data object, since the initial input data object has already been processed. The operation receives *NULL* as input data object when it is restarted after displacement of its thread. Any useful information from the input data object needs to be kept in the operation class members. The last important point to be careful with is the state that is stored in these member variables. Since information associated to operations is only serialized when they are suspended, it is necessary to ensure that the state is consistent when the operation gets suspended. The only points where an operation can be suspended is when it calls the DPS methods *postDataObject* and *waitForNextDataObject*. The requirements described above concern only those operations that are running on threads within malleable thread collections.

The following example shows a restartable split operation. In this example, the variable *i* used as a counter in the loop is incremented before the call to *postDataObject* in order to ensure that when the operation is restarted, the counter already is at the correct value for the next iteration. The initial value of the counter is only set when the operation is called with a non-NULL input data object, i.e. when it is not being restarted.

```
class Split : public dps::SplitOperation<InData,OutData>
{
  CLASSDEF(Split)                  // Define persistent operation state
    BASECLASS(dps::OperationBase)
  MEMBERS
    // Current item (loop counter)
    ITEM(Int32,i)
    // Number of items to output (loop bound from initial input)
    ITEM(Int32,count)
  CLASSEND;

public:
  void execute(InData *in)
  {
    // The input data object is valid (not NULL) if the operation is not
    // being restarted, variables are only initialized in that case
    // (otherwise the variables have already been restored from the
    // checkpoint).
    if(in != NULL)
    {
      i=0;
      count=in->count; // Number of items copied from input data object
    }

    // Loop until all output data objects have been generated
    while(i<count)
    {
      OutData *o = new OutData(i,count);
      i++;   // Pointer must be incremented before posting to ensure
             // that loop restarts from next iteration.
      postDataObject(o);
    }
  }
};
```

5.5 Data type reflection and serialization

During the execution of parallel schedules, the DPS library needs to send tokens and their associated data objects from one node to another. The performance of these transfers is critical to the overall performance of the system. It is therefore necessary to provide a high-performance mechanism for serializing and deserializing the user-defined data objects when they need to be transmitted from one node to another.

Many modern object oriented languages, such as Java [Gosling96] or C# [Petzold01] provide reflection and serialization mechanisms in their respective compilers. However, since C++ does not provide any reflection or serialization mechanisms, DPS needs to provide its own mechanism for accomplishing these tasks. The mechanism should not be too cumbersome to use for the application developer, should provide high performance, and should also support heterogeneous clusters.

5.5.1 Related work

A common approach for implementing serialization in C++ libraries is to provide input and output stream classes, and to have the developer write specific serialization routines to read or write data structures respectively from or to a stream. This is for instance the case when using the standard C++ streams [Teale93] (by overloading the left and right shift operators), or when using the MFC library [Prosise99]. Some libraries provide collections of macros which are used to generate serialization functions [Attardi01]. All of these approaches require multiple definitions of the internal structure of the object, once for the usual C++ class declaration, and at least once for the serialization code. Since all these definitions must match in order to achieve correct behavior, these approaches are cumbersome to use and error-prone.

Metaobject protocols, such as OpenC++ [Chiba95], and language extensions, such as the E language [Richardson93], add reflection and persistency to the C++ language. These approaches require specific compilers or preprocessors to handle the source code. While simple to use and usually very efficient, the additional tools required to build parallel and distributed applications may severely limit their potential audience.

The popular MPI parallel programming library also uses a duplicate definition of data structures that is passed to its send and receive functions. Libraries have been designed on top of MPI in order to simplify communications by automatically creating MPI data types from user-defined complex data types. However, they rely either on a preprocessor [Goujon98] or are restricted to specific type libraries such as the Standard Template Library (STL) [LieQuan03].

5.5.2 Reflection and serialization in DPS

For DPS, we propose an approach based on C++ templates and macros that provide full reflection and serialization capabilities, without requiring any duplicate definition of the data and without adding any memory or execution time overhead to the objects. The approach we propose enables the serialization of any C++ object, whether it is an instance of a simple class, or an instance of a complex templated derived class. It does not require any redundant information within the application source code. Only a few macros are used to package the class members. Virtual methods are not added to classes unless these are necessary for polymorphic constructs. Thus, in the general case, our approach alters neither an object's memory footprint nor its behavior.

The serializer is composed of three layers. The first layer is a mechanism for providing data type reflection within classes, allowing to enumerate individual elements within a class, and to perform actions on these elements (such as read and write). The second layer is a converter that converts from simple types to the desired output format (e.g. to a binary stream or to an XML stream). The third layer is the stream interface that outputs or reads the resulting stream to or from a device (e.g. a file or a network socket).

Since DPS can run applications on heterogeneous clusters, the serializer needs to provide support for changes in endianness, memory alignment and pointer sizes when data is transferred from one node to another. Time consuming tasks such as endian flipping are only performed when required; there is no predefined format for serialized binary data.

5.5.3 Data type reflection for serialization

In order to automatically serialize a data object, it is necessary to enumerate all the elements present in a data structure, providing their name and type, as well as getting a list of all the object's base classes. This information is provided by a data type reflection model. For example, for the following class *Foo*, the reflection model needs to return the information shown in Figure 83.

```
class Foo : public Bar
{
  int a;
  int b;
  std::vector<std::string> strings;
  int c[32];
};
```

	Type	Name
Base classes:	Bar	
Data members:	int	a
	int	b
	std::vector<std::string>	strings
	int[32]	c

Figure 83. Reflected content of a simple class

Reflection information cannot be obtained from the code above when using a standard C++ compiler. Creating this list requires additional elements in the source code, which however should not require the duplication of data type definitions. We propose the following syntax for the class definitions, wrapping every class member within a macro:

```
class Foo : public Bar
{
  CLASSDEF(Foo)
    BASECLASS(Bar)
  MEMBERS
    ITEM(int,a)
    ITEM(int,b)
    ITEM(std::vector<std::string>,strings)
    ARRAY(int,c,32)
  CLASSEND;
};
```

The initial *CLASSDEF* indicates the beginning of the list of base classes. *CLASSDEF* is followed by zero or more *BASECLASS* declarations, indicating the base classes of the defined class. The list of base classes is used to ensure that the contents of the base classes are properly serialized together with the contents of the class itself. The list of base classes is concluded with a *MEMBERS* declaration, indicating the beginning of the list of class members. Each class member is subsequently declared wrapped in an *ITEM* macro. The complete declaration is terminated with *CLASSEND*. This syntax duplicates only the list of base classes and the class name, since they cannot be deduced from any element within the class.

Variants of the *ITEM* macro take care of the various member variable access declarations available in C++.

Macro	Equivalent to
`ITEM(type,name)` `PUBLICITEM(type,name)`	`public: type name;`
`PRIVATEITEM(type,name)`	`private: type name;`
`PROTECTEDITEM(type,name)`	`protected: type name;`
`ARRAY(type,name,size)` `PUBLICARRAY(type,name,size)`	`public: type name[size];`
`PRIVATEARRAY(type,name,size)`	`private: type name[size];`
`PROTECTEDARRAY(type,name,size)`	`protected: type name[size];`

The separate declarations for arrays are required since the size of an array appears after its name rather than together with the type name. Almost any type can be used within the *ITEM* macros, since the DPS serialization mechanism provides support for all basic C++ types and STL containers [Plauger00]. All user-defined types are expected to follow this declaration mechanism, and can be nested arbitrarily. The only types that are not directly supported are pointers, since C++ provides no method to determine how many items are pointed to. This ambiguity makes them unsuitable for direct serialization. DPS provides a mechanism for supporting pointers through the *dps::SingleRef* class, which can be used to wrap a pointer. This wrapper has two functions: first, through its name, inform the developer that the pointer is expected to point to a single item, and second to provide functionality for object lifetime management. The object lifetime management features of DPS are presented in section 5.6.1.

The modified class should be identical to the original, i.e. the macros should not add any data, should not require the modification of constructors or destructors, and should not add any virtual functions. The macros are therefore allowed to produce only simple member functions, since new data members or virtual member functions would alter the footprint of the object.

The resulting reflection information for the class is stored in type lists similar to those presented for the compatible data object type lists in section 5.2.5. The internals of the mechanism are presented in Appendix C.

5.5.4 Data formatting

The type list that is created by the previously described macros is used to generate the required code for serializing and deserializing the data object. The first step in serialization is to format the data. DPS can use both binary and text formats. The binary format is trivial: DPS performs a simple memory copy of the individual fields, except in some particular cases where the binary format is not compatible across heterogeneous platforms. This includes in particular the built-in C++ *bool* type, which does not have the same size on all platforms. Another important aspect that has to be considered when using binary formats is the byte order (endianness) of the data. In order to optimize performance, DPS always sends data using the byte order of the sending platform. If the receiving platform does not use the same byte order, it performs the required byte exchanges on all received data. This approach is more efficient than specifying a fixed byte order for all communications as is for example the case in Java. The text format converts all individual fields into text equivalents, and also provides the names of the reflected fields.

The formatted data is forwarded to a stream, which is ultimately responsible for sending the data to the appropriate destination or for receiving data. DPS provides streams for transferring data over TCP sockets, which are the primary communications mechanism of the library. Other streams are also included, in particular for writing and reading data to and from files. These streams are however used only for debugging. The details of the data formatting classes and the streams are provided in Appendix C.

5.6 Runtime functionality

Beyond the aspects directly related to the execution of parallel schedules, the DPS library needs to provide fundamental services used both internally by the library itself and by user applications. Some of these services, such as the replicated data handler and the class factory, have already been presented in the previous sections. The present section describes further services that have an impact on the library.

5.6.1 Object lifetime management

Many tasks within the DPS library require passing of objects from one part of the library to another, from the library to user-provided functions, and from user-provided functions to the library. Since DPS is heavily multithreaded, many of these transfers involve passing objects from one thread to another, and often also from one thread to several other threads. This makes simple passing of object ownership impossible to use, since it is not possible to assign one single owner to any object. Therefore, DPS needs a garbage collection mechanism [Wilson92].

Since the C++ language does not provide any internal support for garbage collection, the DPS library needs to provide a mechanism to handle the lifetime of objects. A popular solution for garbage collection in C and C++ applications is the conservative Boehm garbage collector [Boehm93]. This is however a very general solution and DPS can use a simpler, more specific approach. Not all the objects in DPS need to have lifetime control. Lifetime control is required only by the objects that are used in asynchronous operations and passed among multiple threads. The objects that are managed by the replicated data handler already have a mechanism for handling their lifetime, since the replicated data handler maintains a distributed reference count on the objects. Therefore, using simple reference counting on the remaining objects is sufficient to ensure that they are properly destroyed when they are of no more use.

In order to keep the reference counting as simple as possible, those objects that require it have a reference count embedded in their data structure. The reference count is updated with the simple interlocked increment and decrement instructions provided by most processor architectures in order to ensure safe updates in multithreaded, multiprocessor environments. When the reference count of an object reaches zero, it is immediately destroyed. DPS provides two methods, *addRef* and *release*, for updating the reference counter.

Developers using DPS will usually not encounter the mechanism, since all data objects passed into operations are destroyed by DPS when the operation terminates, and the data objects created by the operations are destroyed either when they are passed to the network or locally processed in another operation. One of the cases of interest to application developers is for example when an operation wishes to output its input data object without creating a copy. In this case the reference count needs to be incremented in order to avoid having the object destroyed when the operation terminates, since the object needs to persist until it has been passed on to the next operation. The following example shows how a leaf operation can post its input data object:

```
class Process : public dps::LeafOperation<DataType, DataType>
{
  void execute(DataType *in)
  {
    // Perform in-place processing on input data object 'in' (...)

    in->addRef();         // Increment reference count
    postDataObject(in);  // Post in as output data object
  }
  IDENTIFY(Process);
};
```

5.6.2 Remote procedure calls

DPS provides a simple remote procedure call mechanism that is used internally for various simple tasks, such as retrieving remote flow graph sections and requesting new application executions from the DPS kernel. The RPC mechanism is closely tied to the serialization capabilities and the network layer. A standard C function can easily be transformed into a remotely callable procedure by ensuring that it takes one serializable object as argument, and returns another serializable object.

Remote procedures are called through a call handler which supports both synchronous and asynchronous RPC function invocations. Synchronous invocations block until the result of the function is returned, while asynchronous invocations do not block and use a callback function to notify the caller that the function has completed. The call handler automatically determines which operation should be invoked based on the data object type that is passed as argument. The following source code shows an example of a synchronous RPC function call:

```
// Simple example of a synchronous RPC function
MyInputData *in = new MyInputData ( /* constructor args */ );

// Call RPC handler with an object of type MyInputData; this will
// call the associated function on the remote node
MyOutputData *out = (MyOutputData *) getRPCHandler()->call(host, in);
```

On the remote side, the remote procedure is declared as a standard C function, taking an appropriate input data object as argument, and returning another data object. The function is registered with the remote procedure call handler with the *REGISTERRPCFUNC* macro. The registration is performed using a similar mechanism to that of the class factory. The RPC function is subsequently identified by its input data object type. Whenever a data object of that type is received over the network, the RPC function is invoked. The called functions may return both synchronously or asynchronously. The example below shows a synchronous return, where the function immediately returns a value. Asynchronous functions use the information passed in the context (the *RPCContext* passed as an argument) which provides a callback function to use for the asynchronous return, as well as useful information such as the origin of the call.

```
// Simple example of a synchronous RPC function
MyOutputData *myRPCFunction(RPCContext *ctx, MyInputData *arg)
{
  MyOutputData *retVal = new MyOutputData();
  // Do something here…
  return retVal;
}
REGISTERRPCFUNC(myRPCFunction,MyInputData)
```

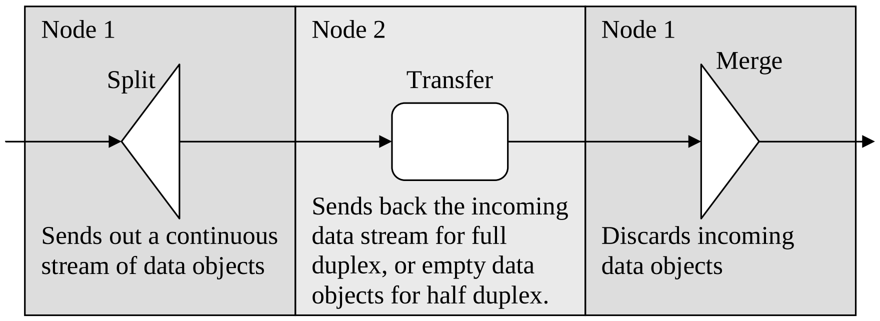
In order to handle failures, the RPC mechanism uses a timeout. This timeout is applied both to the call as a whole from the client side, and to the duration of the call on the server side.

5.7 Low-level performance

The previous sections presented various internal elements of the DPS framework. The performance of these elements is critical to the final execution performance of the complete parallel application. We present in the following sections performance measurements for the serialization and networking mechanism, as well as for the executor subsystem.

5.7.1 Serialization and networking

The first test performed was to use a simple flow graph in order to evaluate the maximum usable network bandwidth in full duplex and half duplex cases when using DPS. The flow graph is illustrated in Figure 84. The parallel schedule is running on two nodes, with the intermediate transfer operation either sending back the incoming data for full duplex transfers, or discarding the data for half duplex transfers.



Node 1
Split
Sends out a continuous stream of data objects
Node 2
Transfer
Sends back the incoming data stream for full duplex, or empty data objects for half duplex.
Node 1
Merge
Discards incoming data objects

Figure 84. Flow graph for measuring network performance

Since we want to measure the sustained network throughput, the parallel schedule needs to run for enough time to ensure that the system enters a steady state. In order to avoid creating huge queues of data objects the flow graph uses flow control, thereby limiting the total memory consumption. The measurements were performed on the cluster of dual processor Pentium III systems in order to obtain results for both Fast Ethernet and Gigabit Ethernet. The results are shown in Figure 85.

The graph indicates the effective useful bandwidth, i.e. the number of data bytes that were stored in the application-defined data object. Since DPS sends some additional information in the tokens used to package the data objects, and TCP/IP also adds further overhead, the effective amount of data sent over the network is higher. The half-duplex case does not represent true half-duplex communications, since the transfer operation always returns a small data object for each incoming data object.

As expected, the Fast Ethernet levels out around 11 MByte/s for half duplex, and at nearly 20 MByte/s for full duplex, reflecting the maximum bandwidth that can be reached on this type of network. The Gigabit Ethernet can reach higher bandwidths than the Fast Ethernet.

We may also examine the number of transfers that can be performed per second, shown in Figure 86. The number of transfers is the total number of incoming and outgoing data objects in full duplex communication. With data objects containing up to 1 kByte of data, the number of transfers per second stays approximately constant at around 13000, and does not significantly differ for Fast Ethernet and Gigabit Ethernet, indicating that the network transfers are mostly limited by the available processing power and the network protocols. Network latency does not play a major role here since these are not ping-pong transfers but simple bidirectional data streaming. As the data objects get larger, the influence of the different bandwidths of the networks can be seen.

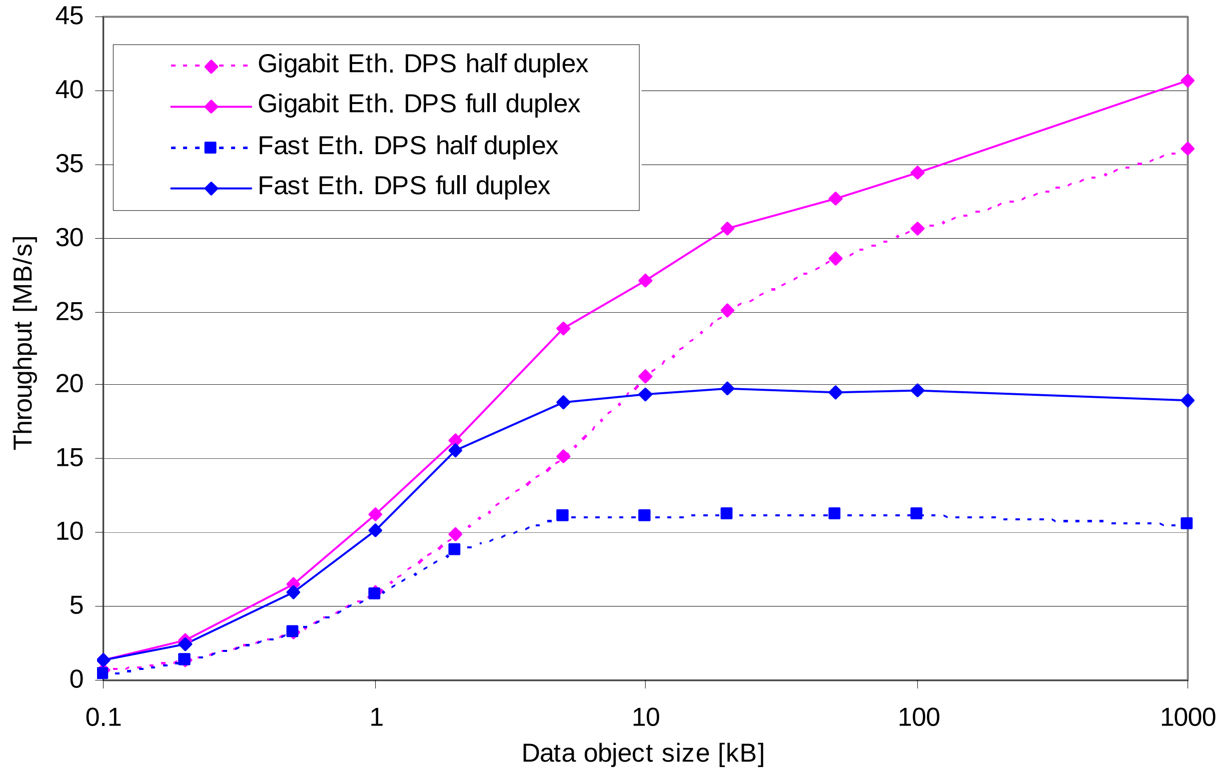


Figure 85. DPS network bandwidth for full and half duplex transmissions on Fast Ethernet and Gigabit Ethernet on the cluster of Pentium III PCs

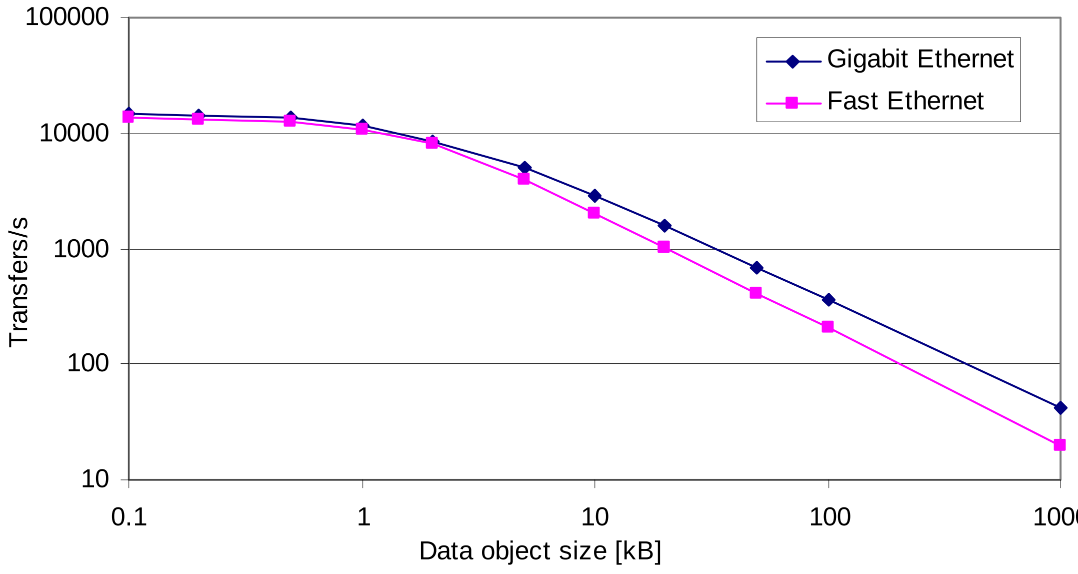


Figure 86. Number of full duplex data object transfers per second based on DPS data object size on the cluster of Pentium III PCs

Finally, we compare the network throughput obtained with DPS with the performance of simple raw sockets. As soon as the data object size reaches approximately 5 kBytes, there is no significant difference in performance. For smaller data object sizes, DPS is slower, since it incurs additional overheads due to the additional token information, and the fact that the data object contains a dynamic buffer the size of which is not known in advance. Therefore DPS needs at least three read operations in order to receive the complete data object: reading the token, the size of the data buffer, and the data buffer itself. DPS also requires multiple memory allocations for the token, the data object, and the data buffer within the data object. For the raw throughput, none of these allocations is required, and the data block can be read in a single operation since its size is known in advance.

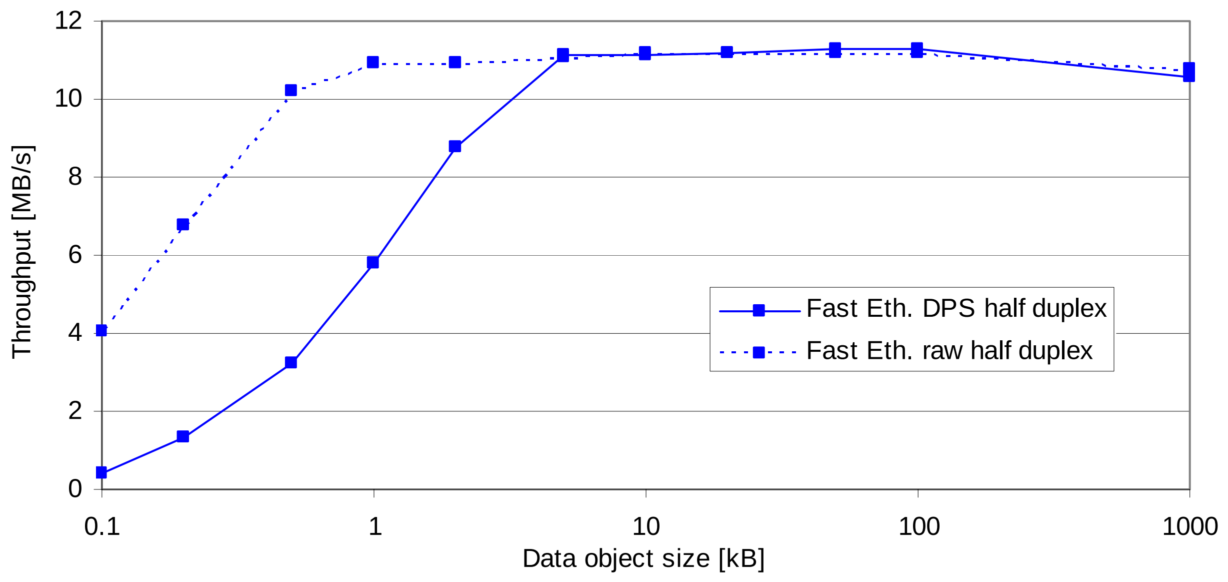


Figure 87. Comparison of DPS half duplex transfer rates with simple TCP sockets on the cluster of Pentium III PCs

5.7.2 Executor performance

The executors are responsible for executing all the operations within a DPS application. It is therefore important to ensure that the sequencing of operations within the executor is performed as efficiently as possible. The first important point is to ensure that the executor does not consume too much time for preparing the next operation after the termination of the previous operation. On the dual processor Pentium III machines, a simple flow graph with a single loop around an operation was run to evaluate the operation to operation time of DPS executors. The flow graph is illustrated in Figure 88.

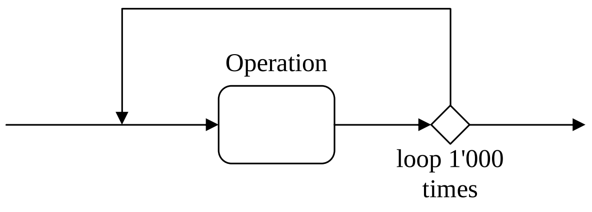


Figure 88. Flow graph for measuring operation to operation latencies

The data object circulating inside the graph stores a timestamp, and every time it reaches the operation, the operation evaluates the time difference between the current time and the stored timestamp. The timestamp within the data object is subsequently updated, and the data object is posted as output. The operation is mapped to a single thread on a single node in order to remove any network communications and capture only the overhead of the framework.

The complete round-trip time for the data object is 79.7 µs. This complete time can be split into several distinct components: the posting of the output data object and its storage in the executor's queue, the destruction of the operation, the internal processing of the data object within the executor in order to find out which operation should be executed next, and finally the construction of the new operation followed by its invocation. The distribution of this time among the various components is shown in Figure 89.

The longest part is the construction of the new operation, and the detail of its internal time consumption is shown in Figure 90. The longest single component in the round trip is the creation of the stack frame for the execution of the operation, requiring 14.7 µs. This time is spent entirely in a single call to the Windows API function *CreateFiber*. The initial call to *SwitchToFiber* to start the execution is also particularly slow (6 µs), but subsequent calls are much faster (1 µs).

The next largest consumed time is due to the calls to the system memory heap. In our round trip, both the operation and a token are allocated and subsequently freed. The various STL containers used by the DPS library may also perform memory allocation operations. A typical operation on the memory heap takes around 3.5 µs.

Another time consuming task is the cumulative time spent in operating-system provided synchronization primitives, such as *EnterCriticalSection* and *LeaveCriticalSection*. Although entering and leaving a single critical section when there is non contention is very fast (0.125 µs), DPS often uses fine grained locking in order to provide as much asynchronous execution potential as possible. Other lock types used by DPS such as read/write locks are also slower than simple critical section accesses, since their implementation requires the use of multiple synchronization primitives.

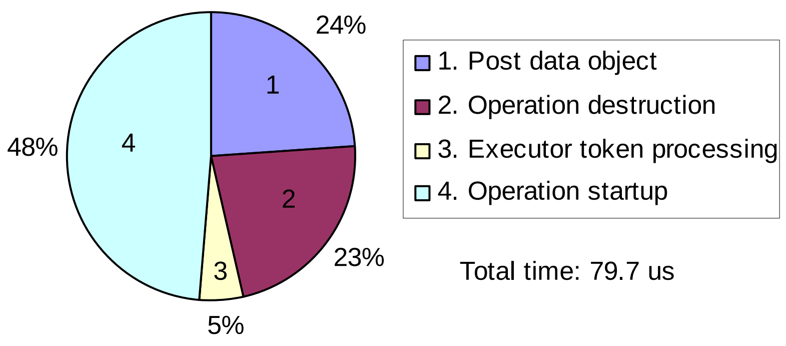


Figure 89. Time distribution for the round trip time from operation to operation

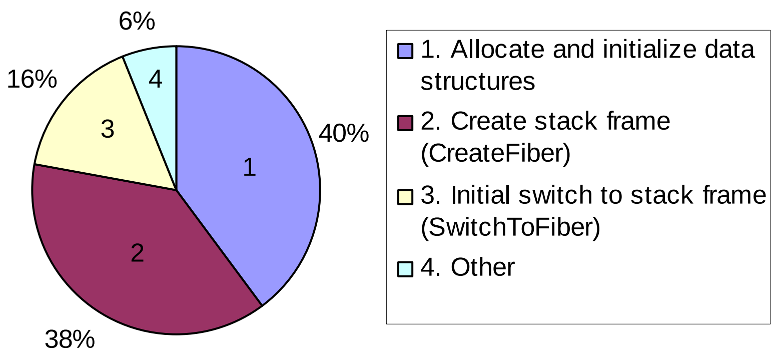


Figure 90. Time distribution for the startup of an operation

This analysis of the executor behavior gives some hints for possible optimizations of the DPS framework in order to minimize internal latencies. The most interesting avenues are the re-use of stack frames, operations, and tokens. Currently, all these items are created and destroyed as required, but they could potentially be stored in pools and be re-used as necessary. Efficient resource re-use could improve the internal executor performance by approximately 30%.

5.7.3 Network latency

The previous section studied the operation to operation latency within an executor running on a single machine. Another critical latency that needs to be minimized is the operation to operation latency when moving from one compute node to another. This latency is equivalent to the previously measured latency plus the time required to send a data object from one compute node to another. In order to evaluate this latency, we have extended the previous flow graph as illustrated in Figure 91.

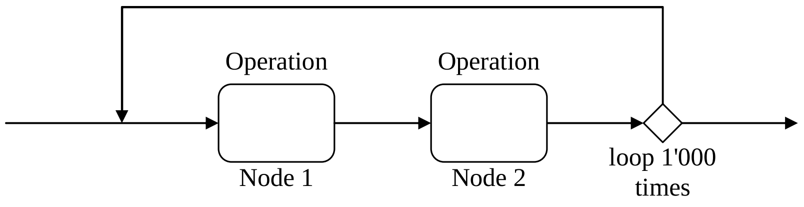


Figure 91. Flow graph for measuring operation to operation latencies with network communications

As before, the data object circulating inside the graph stores a timestamp, and every time it reaches the operation running on node 1, the operation prints out the time difference between the current time and the stored timestamp. The timestamp within the data object is subsequently updated, and the data object is posted as output. The operation running on node 2 simply sends the incoming data object back to node 1.

The complete round-trip time for the data object is 545 μs when running on Gigabit Ethernet, and 600 μs when running on Fast Ethernet. Since this round trip time comprises two operation executions and two network transfers, we can infer the duration of one network transfer to be 193 μs on Gigabit Ethernet and 220 μs on Fast Ethernet.

5.7.4 Operating system scheduler influence

Since DPS is a heavily multithreaded library, the efficiency of the operating systems scheduler plays a major role in the effective application performance. In order to maximize scheduling efficiency, DPS dynamically adjusts the priorities of its various threads depending on their current activity in order to ensure that the most urgent tasks get executed first. The effect of this dynamic priority adjustment can be seen in Figure 92, where the game of life is executed on the cluster of dual-processor Pentium III machines, once with the dynamically adjusted thread priorities, and once without. In this particular example, without the dynamic priority adjustment, most iterations take nearly twice as long as expected.

The delay is caused by some data objects not reaching the networking layer before their thread is pre-empted by the operating system, which then gives control to one of the computation threads. The data object only reaches the network when the computation is complete, thereby delaying the whole iteration by the computation time. This phenomenon only takes place when the computations are fairly short, and therefore the computation thread is not pre-empted by the operating system. These additional delays can however influence long computations, by accumulating the delays caused by the short operations which handle many data objects, especially the split and merge operations present in all parallel schedules.

The thread priorities in DPS are assigned by classifying all threads into one of three categories: executor, network, and internal. The executor threads are all the threads that are bound to the DPS executors used for executing parallel schedule operations. The network threads are all the threads that perform communications on TCP/IP sockets. The internal threads are all the threads that are used internally by DPS for managing the replicated data and other tasks, and are not usually active during the parallel schedule execution. The network and internal threads have a constant priority, with the network threads having a higher priority than the internal threads. The executor threads have by default the same low priority as the internal threads. Executor threads that are currently processing an operation which is likely to post data objects have their priority boosted beyond the priority of the network threads. When the operation posts a data object, it is placed in the network thread's queue. The executor thread's priority is immediately lowered again, leaving the network thread with the higher priority. The network thread which has the data object in its queue usually gains the next execution timeslice, since it is the highest priority thread that is not suspended on a synchronization primitive.

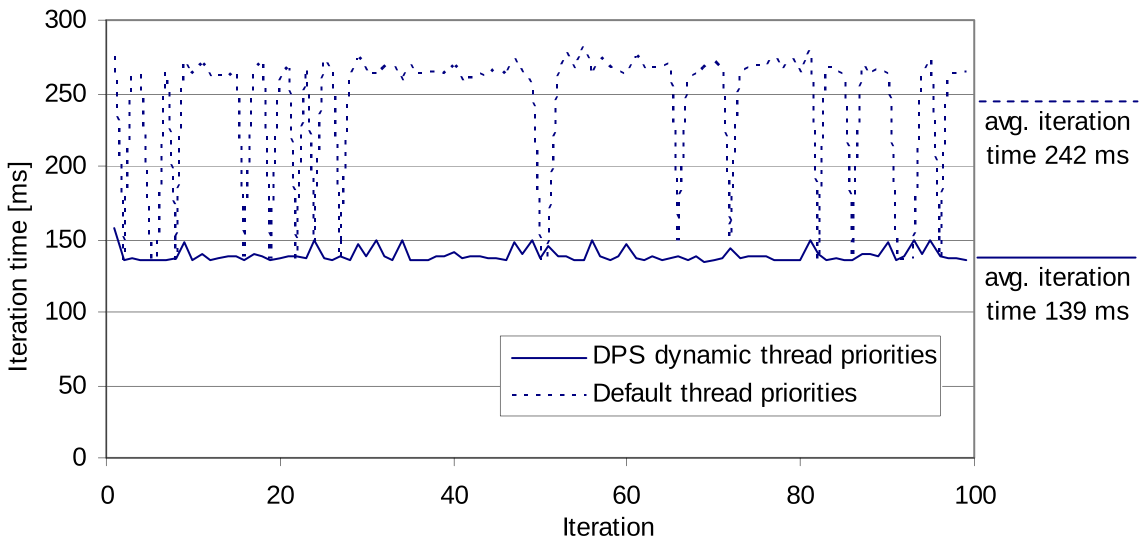


Figure 92. Iteration times for the game of life with and without DPS thread priority adjustment

5.8 Conclusion

This chapter presented the internal object model and execution model of DPS. The DPS object model is based on a collection of template classes that are exposed to the developer, providing strict typing and compile-time validation. Below these external template classes, DPS uses a typeless internal object model capable of being transferred over the network, even to applications that do not have implementations of all the involved types. Most internal objects are managed by a replicated data handler, which is responsible for ensuring that the internal data structures are available on all nodes participating in the parallel computation and that their state is consistent.

The execution of parallel schedules is based on queues of data objects that are processed asynchronously on distinct operating system threads. These data objects are packaged inside tokens that provide sufficient information for the DPS framework to determine the complete execution context required for the execution of operations. In order to enable the use of standard C looping constructs in split operations and merge operations, DPS uses stack switching in order to enable multiple operations to coexist within the same operating system thread.

When a data object needs to be transferred from one node to another, DPS uses a built-in serialization mechanism to generate a binary representation of the data object that can be streamed over the network. The object can subsequently be reconstructed by the receiver. This mechanism accounts for heterogeneous platforms, in particular by taking care of byte ordering and alignment issues.

In order to achieve asynchronous execution, all internal functions in DPS are multithreaded, thereby ensuring maximum utilization of the available processing resources. Individual operating system threads are used for executors, for network transfers, and for many internal tasks such as updating replicated objects. The priorities of execution threads are dynamically adjusted in order to ensure maximum execution efficiency.

The internal object model is very flexible, allowing for the implementation of all the dynamic features required by the specification of dynamic parallel schedules. This flexibility also provides the base for implementing fault tolerance, which is the subject of the next chapter.

Chapter 6

Fault Tolerance with Dynamic Parallel Schedules

Parallel programming environments need to provide support for fault tolerance, since the ever larger number of nodes present in commodity clusters no longer allows guaranteeing failure-free execution. DPS facilitates the implementation of efficient and transparent fault tolerance thanks to its flow graph based application descriptions and dynamic remapping capabilities. This chapter presents the mechanisms that were added to DPS in order to enable fault tolerance. We quantify the impact of fault tolerance on application performance both theoretically and experimentally.

6.1 Introduction

Large clusters of commodity computers such as those for which DPS is designed are not highly reliable systems. Failures occur due to hardware problems, or even due to simple user intervention, e.g. when the computing nodes are accessible to students. It is therefore essential to provide mechanisms within the DPS framework enabling applications to pursue their execution despite the presence of single or multiple node failures.

This chapter is divided into two sections. The first section presents the fault tolerance mechanism and its inner workings. The second section shows the performance impact of the fault-tolerance mechanism on application performance. Parts of the results presented in this chapter have been published in [Gerlach05].

6.1.1 Related work

DPS applications communicate by passing messages over a network. In the context of message-passing systems, two major classes of recovery schemes have been proposed: checkpoint-based and message log-based recovery [Elnohazy02].

Checkpoint based approaches store the current state of computation to stable storage. *Coordinated checkpointing* on all participating nodes [Tamir84] may be achieved by stopping in an ordered manner all computations and communications, and performing a two-phase commit in order to create a consistent distributed checkpoint. Checkpointing can also be performed independently on all participating nodes (*uncoordinated* checkpointing). This removes the performance bottlenecks induced by the global synchronization required for coordinated checkpoints and allows checkpointing at convenient times, for example when the data size associated with a checkpoint is very small. Several checkpoints need to be stored on each node, and a consistent state from which to restart has to be found when a failure occurs [Bhargava88]. In unfavorable situations, the recovery can lead to the *domino effect*, where no consistent checkpoint other than the initial state can be found. In order to eliminate the domino effect, additional constraints on checkpointing sequences need to be introduced, for example based on the applications' communication patterns [Wang93].

Message logging approaches store in addition to checkpoints all the messages flowing through the system. The logged messages allow bringing a node to any given state by re-executing its application code with the corresponding sequence of logged input messages. Three types of message logging are usually considered: pessimistic, optimistic and causal. *Pessimistic* logging logs every received message to stable storage before processing it. This ensures that the log is always up to date, but incurs a performance penalty due to the blocking logging operation. The penalty can be reduced by using specific storage hardware, or by using sender-based message logging [Johnson87][Chakravorty04]. *Optimistic* logging begins processing messages without waiting for a successful write to stable storage [Strom85]. The overhead of pessimistic logging is removed, but several messages might be lost in case of failures. When the system is restarted it must roll back to a previous consistent state on all nodes. Finally, *causal* logging also provides low overhead and limits the backtracking that has to be performed during recovery. It does however require the construction of an antecedence graph for messages, and requires a rather complex recovery scheme [Elnozahy92].

These mechanisms make no assumptions about the internal structure of the applications other than the use of message passing for communications between processes. They are thus very well suited for applications written with general-purpose message passing libraries such as MPI. Most fault-tolerant parallel MPI systems rely on coordinated checkpointing [Stellner96][Agbaria99][Chen97]. Further fault-tolerant implementations of MPI rely on uncoordinated checkpointing either with pessimistic message logging [Batchu01][Louca00][Bosilca02] or with pessimistic sender based message logging [Bouteiller03].

When parallel applications are described using high-level approaches, additional information about the structure of the application is available. For example, task graphs [Das97] or Calypso [Baratloo95] make use of such information for recovering and resuming computation after a failure.

Fault tolerance schemes also vary in the assumptions they make about the number and nature of failures that can be recovered. Placing additional limitations on the recoverable cases may enable significant optimizations when compared to the general case. For example, if the system has never more than one failure at a time, stable storage can be replaced with transfers to neighboring nodes [Plank95].

6.2 Fault tolerance in DPS

The fault-tolerance mechanism implemented in DPS relies on message logging and checkpointing. The flow graph with its associated thread collections is a powerful tool for implementing fault-tolerance. Whereas implementations of fault-tolerance for low level message passing libraries have little or no information about the application, high-level frameworks provide information about the applications execution patterns. This additional information is fully exploited within the DPS framework in order to provide nearly transparent fault tolerance to applications.

The fault tolerance mechanism makes some assumptions about the failures it will need to recover from. In particular, we assume that failures are fail-stop failures [Schlichting83], where failed nodes simply disappear from the cluster. We do not consider the possibility of software failures where a node suddenly starts sending erroneous tokens or intermittent failures. Since DPS is a high-level framework, we assume that a partial failure on a given node will lead to that node's shutdown by the underlying operating system.

Another assumption that DPS makes about the application is that it is deterministic. This assumption implies that all operations within the application's flow graph systematically exhibit the same behavior for the same input conditions (i.e. they will post the same output data objects in the same order and apply the same modifications to the local thread state). These limitations also apply to the application's routing functions and loop conditions. These limitations are relaxed when considering leaf operations that do not modify any thread. For these operations, the routing may be non-deterministic, for example by using the built-in load balanced routing.

The fault tolerance mechanism in DPS is based on *backup threads*. For each thread in a thread collection, a backup thread is created. The backup thread takes over the execution of the active thread when the active thread fails. Therefore it is necessary to provide a mechanism allowing the backup thread to continue execution from the same state in which the active thread was when it failed. The backup threads need to be mapped onto nodes that differ from the nodes running the active threads, for example by rotating the thread indices as shown in Figure 93. The backup threads are stored in the system memory of the nodes that host them.

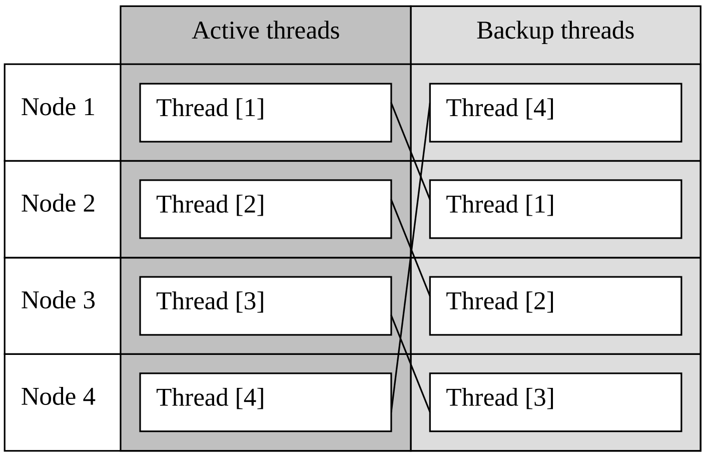


Figure 93. Mapping of active and backup threads onto processing nodes

This type of mapping provides partial support for simultaneous multiple node failures, since as long as for each thread either the active thread or the backup thread survives, the application can pursue its execution. In the example of Figure 93, nodes 2 and 4 could fail simultaneously without compromising the application. In order to support successive failures, the backup threads that were located on a failed node or that have been consumed when replacing a failed active thread are reconstructed on another node.

DPS detects node failures by monitoring communications. A node is considered to be failed when it is not able to communicate with another node. The TCP/IP network layer used by DPS reports failures when communication failures are detected or disconnections occur. When a failure is detected, DPS needs to reconstruct the complete state of all structures that were on the failed node elsewhere. These structures include the internal DPS structures such as the executor's data object queues, as well as the user defined structures such as local thread state data.

6.2.1 State reconstruction

When a backup thread replaces an active thread, the local state in the backup thread is brought up to date by re-executing the operations on a past state of the data structures (either the initial blank state at program startup or a checkpoint). In order to perform this re-execution, the backup thread also needs all the data objects that served as input to the operations. Therefore, when a data object is posted to a thread, it is sent both to the active thread and the corresponding backup thread. The active thread performs the operation, and the backup thread only stores a copy of the data object.

A past thread state and the set of data objects that have been processed are sufficient for a complete reconstruction, since operations in the DPS framework do not have access to any other data structures. Every operation that is executed within the DPS framework has two data input sources: the incoming data object(s), and the local thread state data. Likewise, it has two outputs: the outgoing data object(s), and any modifications made to the local thread state data.

When reconstruction starts, the backup thread contains copies of all data objects and a past thread state. The order of the data objects is the order in which they were received, which does not necessarily represent a valid execution order. It is therefore necessary to add a sequencing mechanism to the data objects in order to allow the reconstitution of a valid execution order for the operations triggered by the data objects.

Since operations are re-executed during the reconstruction process, they will also repost their output data objects. If a data object is sent to an operation on a thread where that operation has already been successfully executed on a previous instance of the same data object, that data object needs to be discarded (silent re-execution property).

6.2.2 Determining a valid execution order

The execution order of operations in a parallel schedule is defined by the flow graph, and constrained by the split/merge operation pairs. Within a split-merge operation pair, the outgoing branches of the split operation may be executed in any order. Let us consider the flow graph shown in Figure 94a. By assuming that the first split operation generates two data objects, and that the second split operation splits each of them into three data objects, we obtain the operation executions shown in Figure 94b. The execution order of the operations is only constrained by the flow graph's directed edges. For instance, two valid execution orders for the graph are {1,2a,3a1,3a2,3a3,4a,2b,3b1,3b2,3b3,4b,5} and {1,2b,3b3,2a,3b2,3a1,3b1,3a3,4b,3a2,4a,5}. Because of the asynchronous execution model of DPS and of external factors such as network load or operating system scheduling, the effective execution order cannot be predicted.

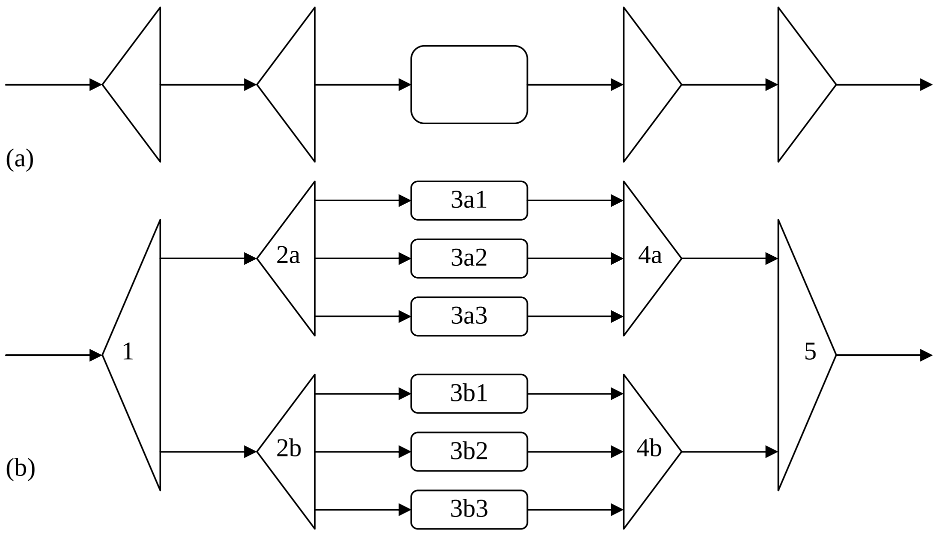


Figure 94. (a) Simple flow graph and (b) operations that are executed when running the parallel schedule (each arrow represents one data object).

According to the flow graph, any data object leaving an operation depends on all data objects having previously entered this operation. In order to reflect this dependency, each operation assigns a sequence number to all the data objects it generates. The sequence number of an outgoing data object is larger than the sequence number of all previously received data objects. We compute the sequence number as follows:

$$SeqOut_{n+1} = \max(SeqIn, SeqOut_n) + 1$$

SeqIn is the collection of sequence numbers of all input data objects that have been received by the operation and $SeqOut_n$ is the sequence number of the previously sent object (if any). This mechanism ensures that the sequence numbers steadily increase. However it does not produce globally unique sequence numbers, since two separate branches of the graph may produce the same numbers. Nevertheless, it does accurately reflect the execution order constraints induced by the DPS flow graph. Figure 95 shows the sequence numbers generated for a neighborhood exchange followed by a computation.

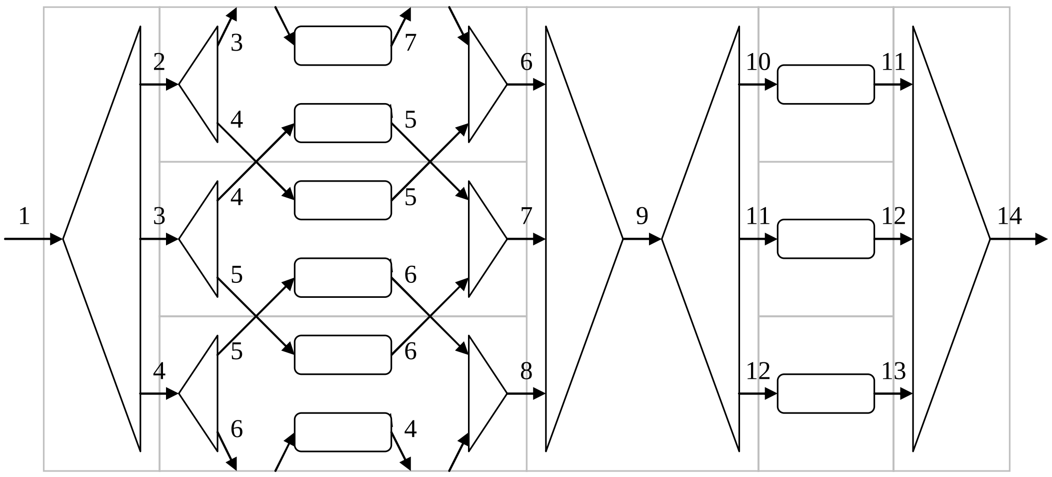


Figure 95. Sequence numbers associated with the transmitted data objects for a neighborhood-dependent computation.

When recovering from a failure and initiating a re-execution on the backup thread's node, the data objects present in the backup thread's queue are sorted according to their sequence numbers. The backup thread is subsequently marked as active, causing execution of operations to resume with a valid execution order. Since the stored data objects contain the operation identifiers, split and merge operations occur with the same combinations of data objects as the original failed execution.

6.2.3 Silent re-execution

Duplicate data objects arising during the re-execution of operations need to be eliminated in order to avoid executing operations twice on the threads that did not fail. In order to detect duplicate data objects, a unique identifier is required for data objects. Since the sequence number is not unique, it cannot be used. However, the hierarchical token identifiers used for assembling data object in merge operations described in section 5.3.3 are unique, and therefore well suited for eliminating duplicate data objects.

The removal of duplicates is performed within the data object queues of the executors. The executor stores the token identifiers of all the data objects it has already processed, and when a data object with the same token identifier appears, it is silently discarded. Since the data object list is pruned whenever checkpointing occurs, its size remains limited.

6.2.4 Checkpointing

In order to shorten the reconstruction time of a failed node, the state of the active threads may be copied onto the corresponding backup threads. Such copies are also needed in order to generate new backup threads when the previous backup threads have been consumed. Each DPS thread has three components that must be conserved for successful reconstruction: the local thread state, the queue of data objects that wait for processing, and the state of active (but possibly suspended) operations within that thread.

The elements that need to be copied for checkpointing are identical to those that need to be copied when a thread is moved from one processing node to another in malleable applications. Therefore, the checkpointing mechanism can be implemented by reusing the elements described in section 5.4. There are however a few differences:

- The queue of data objects waiting for processing does not need to be transferred, since the data objects are already present on the backup thread. They were transferred to the backup thread as part of the fault tolerance mechanism (see section 6.2.1).
- We transfer the list of identifiers of the data objects that have been processed on the active thread since the last checkpoint. The corresponding data objects can be removed from the queue stored in the backup thread, since the state transferred with the checkpoint already contains the results of

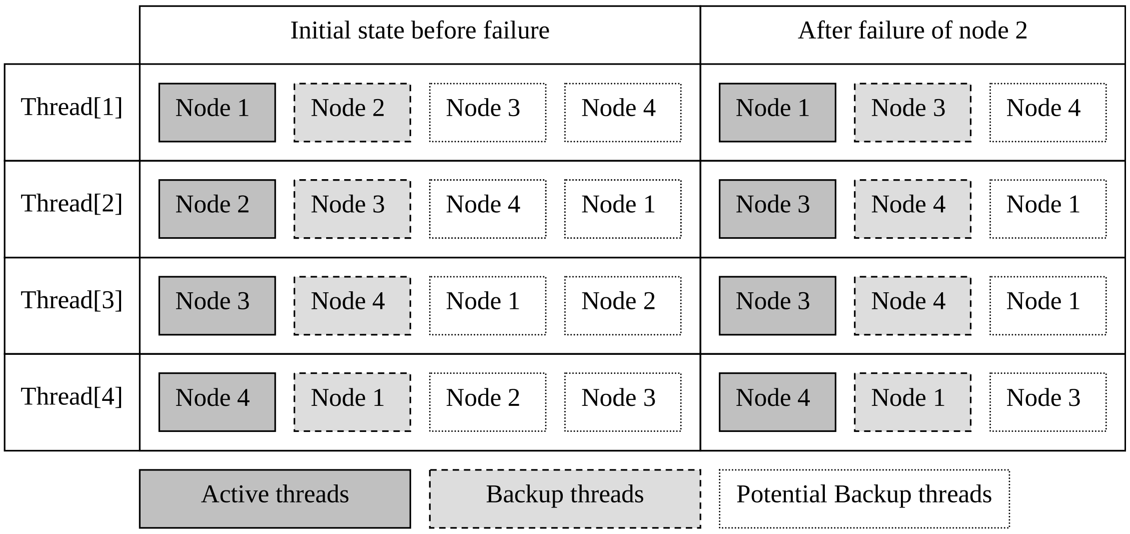
the operations invoked by these data objects. Only the unique identifiers are transferred, not the data objects themselves.

- Taking a checkpoint does not require any updates to the location of threads within the thread collections, since no thread is actually displaced. The active thread continues execution after the checkpoint has been taken, and the backup thread simply stores the new state information.

Since copying the current state also removes part of the pending data object queue on the backup thread, it limits the memory requirements on the backup nodes. This checkpointing operation can be carried out asynchronously and independently on all individual threads. Independent checkpointing of individual threads enables the compute nodes to remain potentially busy during the checkpointing process by executing operations attached to other threads. The effective overhead induced by stopping a thread in order to take a checkpoint can therefore be kept very low.

6.2.5 Successive failures

When a node fails, a certain number of threads lose their active threads or their backup threads. The reconstruction mechanism reconstructs the active threads from the surviving backup threads. The affected threads are then left without a backup thread, making the whole application vulnerable in case of a subsequent failure. DPS provides a mechanism for automatically regenerating backup threads after failures. In addition to specifying nodes for the backup thread for each thread in a collection, a further set of nodes can be specified to use as replacements for the backup thread after failures.



	Initial state before failure				After failure of node 2		
Thread[1]	Node 1	Node 2	Node 3	Node 4	Node 1	Node 3	Node 4
Thread[2]	Node 2	Node 3	Node 4	Node 1	Node 3	Node 4	Node 1
Thread[3]	Node 3	Node 4	Node 1	Node 2	Node 3	Node 4	Node 1
Thread[4]	Node 4	Node 1	Node 2	Node 3	Node 4	Node 1	Node 3

Figure 96. Mapping of threads to nodes before and after failure (active thread, backup thread, potential backups)

When a node fails, it is also removed from the list of potential backup threads. This operation is basically free, since the thread collection data structures need to be updated anyway after a failure, and no data structures are located on the potential backup thread nodes.

When an active thread was located on a failed node and is replaced by its backup thread, the new backup thread is created by making a copy of the previous backup thread before bringing it back up to active status. Performing the copy immediately ensures that the time during which the parallel schedule is fragile is kept to a minimum. When a backup thread was located on a failed node and must be replaced, the active thread is simply checkpointed to create a new backup thread.

The list of potential backup thread nodes can be updated at any time at no cost, since the potential backup thread nodes do not store any specific data structures. It is only important to ensure that the list always contains at least one node, so that a backup thread can be recreated immediately on failures.

6.2.6 Overheads

The proposed recovery scheme induces a communication overhead in running applications, since every data object is sent twice, once to the active thread and once to the backup thread. On applications that are not communication bound, part of this overhead can be hidden, since network transfers are partially overlapped with computations. There is also a memory overhead induced by the backup threads, which contain both a previous thread state and the pending data object queue.

Figure 97 illustrates timelines for the execution of a single iteration of the game of life on a single node with and without fault tolerance. Separate timelines indicate incoming data object communications, computations, and outgoing data object communications. The additional communications induced by the fault tolerance scheme are carried out in parallel with the computation of the center cells and do therefore not significantly affect the parallel program's overall execution time.

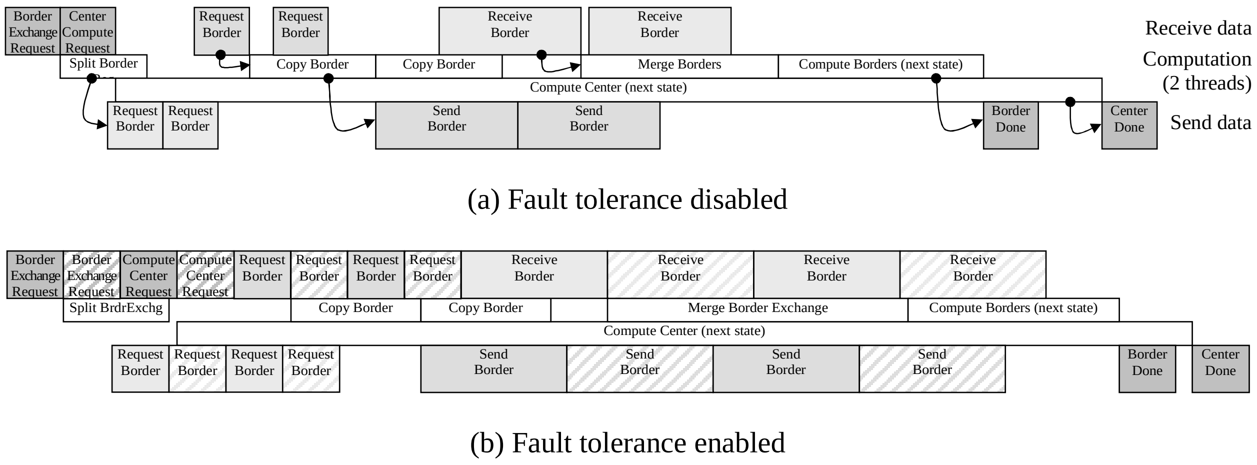


Figure 97. Timelines for a worker node running the improved graph of the game of life, with (a) fault tolerance disabled and (b) fault tolerance enabled (illustrative example, does not reflect real time intervals).

6.3 Recovery without local thread state

The previously described fault tolerance mechanism is designed to handle the general case where threads are used to store local data. However, many applications also use threads that do not store any data for tasks that are composed only of computations. An example for such application behavior is the LU decomposition presented in section 4.3, which uses two thread collections. The first thread collection stores the matrix distributed over all participating nodes, and performs the operations that require access to the matrix data. The other thread collection is used to perform the multiplications, which do not need access to the matrix data stored in the local threads.

In such thread collections of stateless threads, the individual threads are perfectly interchangeable. The only item required to re-execute an operation is its input data object. Therefore, we may remove the backup threads for these types of thread collections, since they store only a data object queue, and store the data objects at their source. When a node fails, all tasks that were executing or queued on the failed node need to be resent to other nodes. Figure 98 illustrates the storage of data objects on the source split operation in a situation where the processing threads are stateless.

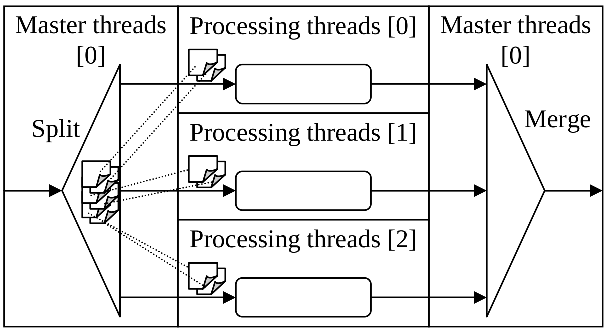


Figure 98. Fault tolerance for stateless threads – store duplicate data objects at the source node

Storing data objects in the split operation is simple, since they are created within that context. The posted data objects are processed by a pipeline of one or more leaf operations running in stateless threads, and the corresponding result is returned to the merge operation. The merge operation subsequently notifies the split operation that the result corresponding to a given data object has arrived, allowing the split operation to remove the locally stored copy of the data object. The notification is based on the DPS flow control mechanism, allowing the number of data objects stored on the split operation to be controlled by the application developer.

When a failure is detected, the thread collection is reconstructed by removing the failed nodes, i.e. by reducing the number of available threads within the thread collection. As long as at least one thread remains within the thread collection, execution of the application can continue. The split operation resends all its stored data objects. Since the thread collection has changed, these data objects are routed to valid computation nodes. All data objects are resent, since their route beyond the first operation is not known to the split operation. An example of such a case is shown in Figure 99.

After a failure, additional threads can be added to the thread collection so as to provide support for subsequent failures. In order to ensure the survival of a parallel schedule, at least the outermost split-merge pair of the application's flow graph must be located on threads with backup threads. Split-merge pairs having backup threads can, in case of failure, be recovered using the general recovery scheme and therefore always initiate the re-execution of the enclosed operations.

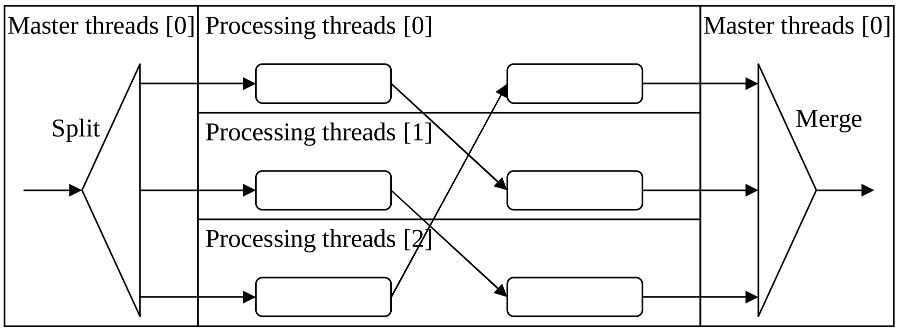


Figure 99. Example of complex routing within a split-merge operation pair

6.3.1 Overheads

The optimized recovery mechanism for stateless threads removes the major overheads of the general purpose mechanism, in particular the requirement to send duplicates of all data objects to the backup threads, and the storage required for the backup threads themselves. These overheads are replaced by the storage requirements for the data objects on the nodes hosting the split operations. During normal execution, the nodes that execute split operations must store the data objects until the corresponding notification from the merge operation is received. This overhead can be limited by using the DPS flow control mechanism, which limits the number of data objects that are simultaneously in circulation.

6.4 Performance evaluation with fault-tolerance

In order to evaluate the performance overheads introduced by the fault tolerance mechanisms, we have developed a simple analytical model for predicting them. This model was subsequently tested with the previously presented game of life and LU decomposition applications.

6.4.1 Prediction model

The performance overhead prediction model only takes into account the additional network communications induced by fault tolerance, since the overhead consists mainly of duplicate messages sent to backup threads and of thread state exchanges between active and backup threads. The following factors need to be considered: the application's communication to computation ratio, the network bandwidth, the relative CPU load for network transfers, and finally the size of the additional network transfers due to fault tolerance. For compute bound applications, the processing time required for the redundant network transfers (duplicate data objects sent to backup nodes) is added to the program execution time observed when fault tolerance is disabled. In communication bound applications, the full network transfer time of duplicate data objects is added. Duplicate data object transfers that take place at the end of the flow graph, such as values returned to the final merge function, can be ignored, since program execution will terminate before the duplicate data objects have been sent to their backup threads.

$$Overhead = \begin{cases} \sum CommTime \cdot CommCPUOverhead & \text{if compute-bound} \\ \sum CommTime & \text{if communication-bound} \end{cases}$$

6.4.2 The game of life

On the game of life, we measure two overheads: the overhead induced by duplicate posting of data objects when fault tolerance is activated, and the overheads induced by checkpointing. These overheads are computed by comparing the execution times of the application with fault tolerance enabled and disabled.

When using large world sizes, the game of life application is compute-bound and has a low communication to computation ratio. In order to demonstrate the influence of the communication to computation ratio, we create a variant of the application by artificially increasing the message size of the border exchange operation. Each node sends an additional 1 MB of data to each of its neighbors, inducing a higher parallel communication load.

The performance measurements were taken on the cluster of Sun Ultra 10 workstations. Performing full-duplex network transfers at maximum speed (9 MB/s in both directions) consumes 50% of the CPU of the Sun workstations. The measurements were performed with both the standard flow graph presented in section 4.2, and the optimized flow graph with the overlapped border exchanges presented in section 4.2.1.

Figure 100 illustrates the overhead in parallel application execution time with the additional border communications. Without additional border communications, the communication to computation ratio is always lower than 0.003 and the corresponding overheads are too low to be measurable. With the additional border communications, Figure 100 shows that the overhead induced by fault tolerance is roughly proportional to the application's communication to computation ratio. The overheads closely match the predictions computed by using the previously described model.

Figure 101 shows that fault tolerance has only a small impact on the overall speedup. The overheads induced by fault tolerance are similar for both the normal and optimized graphs (see section 4.2, Figure 45 and Figure 47) since the additional network load is identical.

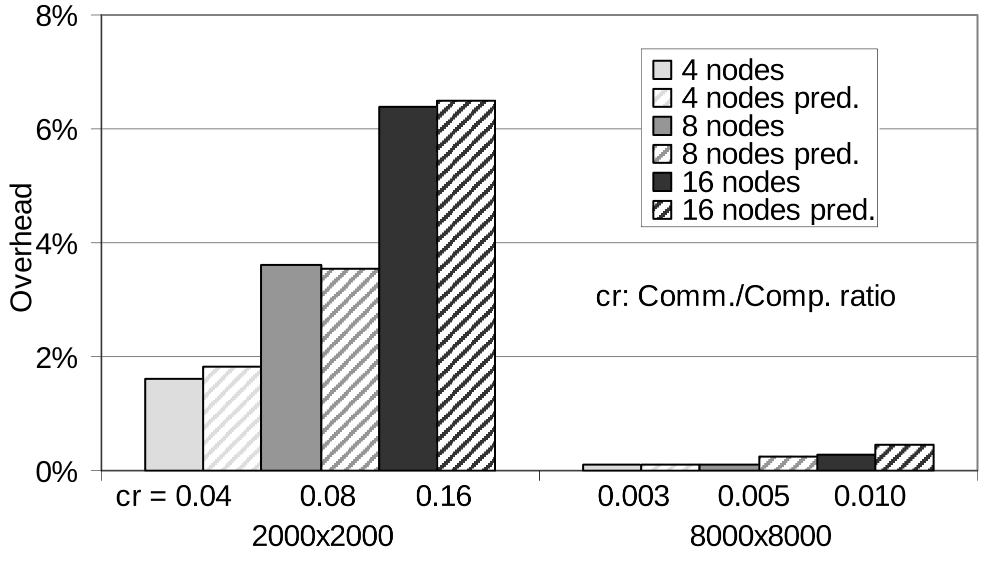


Figure 100. Measured and predicted performance overhead of the game of life when fault tolerance is enabled (optimized graph, 1MB additional border communications)

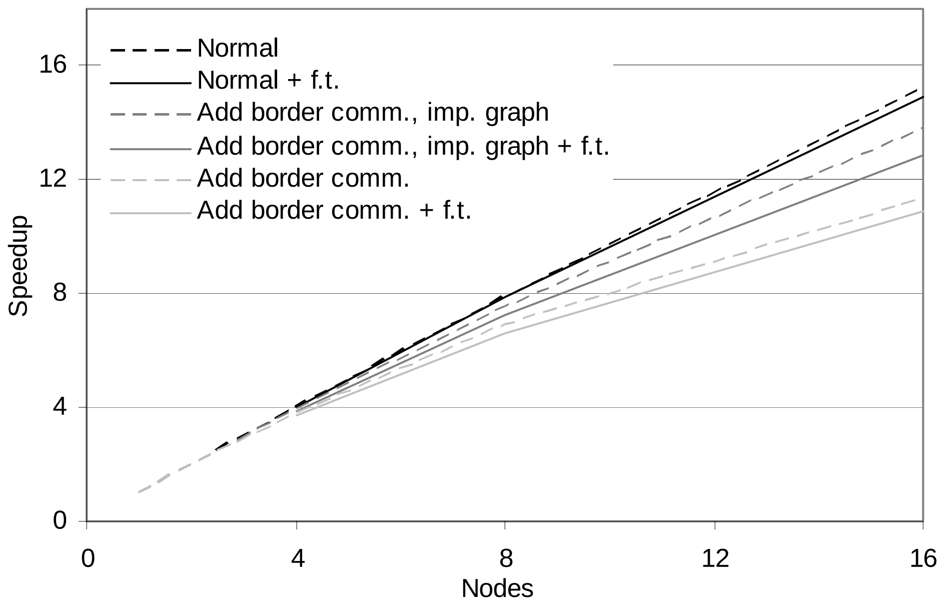


Figure 101. Speedup of the game of life, world size 2000x2000, showing normal execution and execution with additional border communications, with and without fault tolerance enabled (f.t.)

A further overhead induced by fault tolerance is the checkpointing time. The overheads induced by checkpointing can be predicted using the same model as the message duplication overheads, since the checkpoints are transferred to the backup threads during program execution. The only additional overhead is caused by the necessity to lock the individual threads in order to take valid checkpoint images of their states. The overheads for checkpointing were also measured on the game of life sample application, using a 4000x4000 world size and the improved graph. The size of a checkpoint is 32 MB.

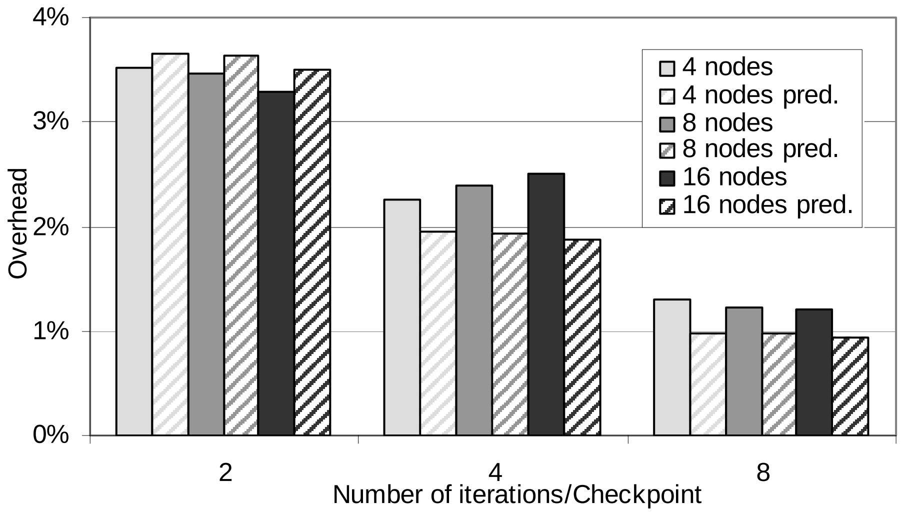


Figure 102. Measured and predicted performance overhead of the game of life for checkpointing, world size 4000x4000.

The checkpointing overhead (Figure 102) shrinks with the number of iterations per checkpoint. The relative overhead is nearly independent of the number of nodes, since state replication is carried out in parallel, and both the size of the replicated state and the program execution time are inversely proportional to the number of nodes.

Finally, let us consider the influence of failures on the execution time of the game of life. Figure 103 shows the total execution time for 100 iterations of the game of life with a world size of 2000x2000 in presence of successive failures. Two sets of measurements were performed, the first with checkpointing enabled, and the second without checkpointing. Checkpoints were taken every 10 iterations. We manually created failures that were roughly evenly spaced along the total duration of the execution. As expected, the checkpointing allows the recovery time to be kept short, at a very slight cost when no failures occur. The total overhead incurred by activating checkpointing when there are no failures is 3.5%. Since the failures are created manually, these measu

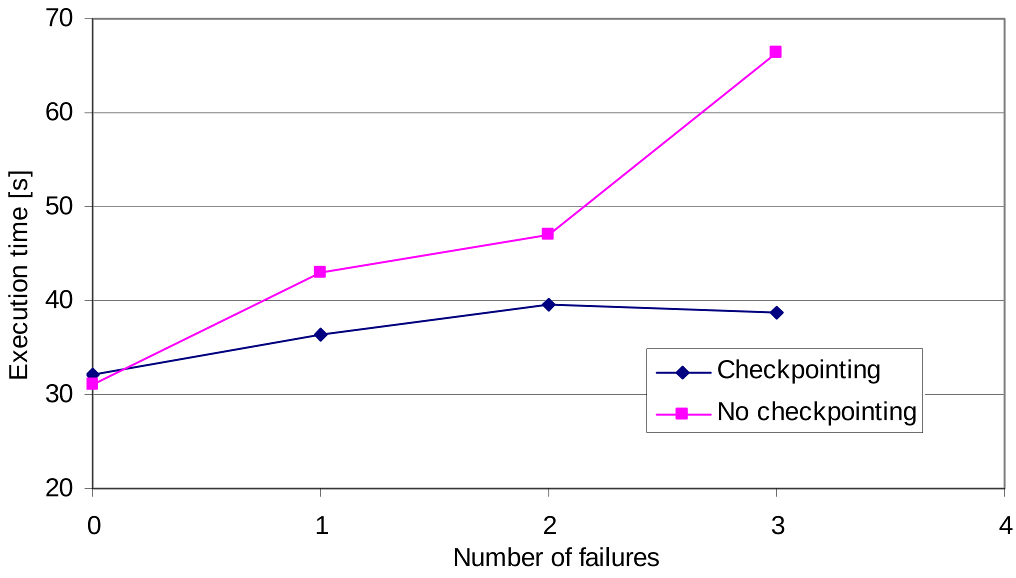


Figure 103. Execution times with recovery in case of successive failures for 100 iterations of the game of life

6.4.3 LU decomposition

With respect to fault tolerance, the LU decomposition is an interesting case to study since it uses two distinct thread collections. One of these collections stores the matrix to decompose in its local thread state, whereas the other collection does not store any data. The application therefore uses both fault tolerance mechanisms. We can force DPS to use the general purpose mechanism for both threads by simply adding state information and backup threads to the multiplication threads, allowing the overhead reduction due to stateless recovery to be quantified.

In order to illustrate the performance overheads, we have executed the LU factorization application on 5 nodes, and varied the matrix size and block size. The results include both the cases with stateless fault tolerance enabled and the cases with stateless fault tolerance disabled, and are shown in Figure 104. For each measurement, the communication to computation ratio is given. The communication to computation ratio can be modified by changing the block size *r* used for the decomposition (smaller blocks yield a higher communication to computation ratio, see section 4.4.4). The block size also influences how deep the application can be pipelined, and the amount of time that is lost waiting for the termination of previous computations.

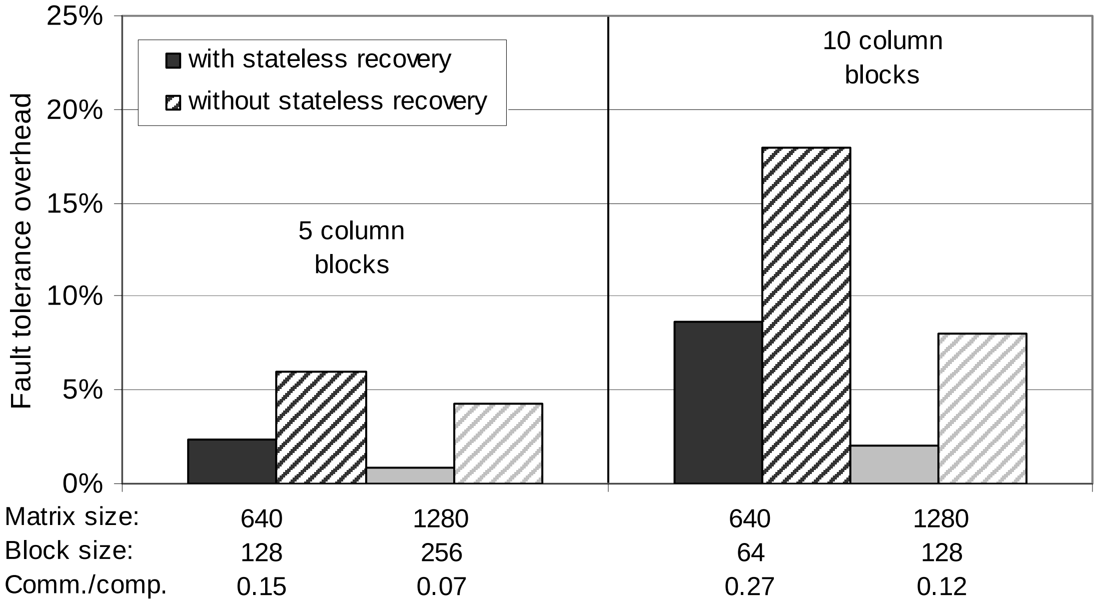


Figure 104. Overheads of fault tolerance for the LU factorization running on 5 nodes with and without the stateless recovery scheme

The cases where the matrix is divided into 10 column blocks instead of 5 achieve a higher parallelization efficiency, with about 25% shorter execution time for the 640x640 matrix, despite having close to twice the communication to computation ratio compared with the case where the matrix is divided into 5 column blocks. The higher communication to computation ratio and also the higher CPU utilization induce the higher overheads. The 1280x1280 matrix divided into 10 column blocks has an even higher increase in efficiency (and thus CPU utilization), but a lower communication to computation ratio than the 640 x 640 matrix, and thus a lower overhead.

A significantly lower overhead is achieved with the optimized stateless fault tolerance mechanism. At high CPU utilization rates (high efficiency), the fault tolerance overhead is reduced by a factor between 2 and 3 when the stateless recovery mechanism is active, since almost two thirds of the transmitted large data objects, i.e. the input matrix blocks, need not be transmitted twice. The output blocks are still transmitted twice since they are sent to the next node of the flow graph which is handled by the general purpose fault tolerance mechanism.

6.5 Conclusion

This chapter presented the extensions of the DPS framework for supporting fault tolerance. Fault tolerance is provided by a hybrid recovery scheme using two compatible mechanisms for the recovery of flow graph program execution segments located on a failed node.

The first general purpose mechanism relies on duplicate data objects sent to backup nodes in order to enable the reconstruction of the state of a thread upon node failure. Backup threads are kept up to date by periodical checkpointing of thread states. Upon occurrence of a failure, the current state of the threads that were on the failed node is reconstructed on the backup threads by reexecuting operations. The valid execution sequence of operations is automatically deduced from the flow graph of the corresponding DPS application by applying a simple sender-based data object numbering scheme. A second specialized sender-based mechanism is used for operations that do not depend on local state information, such as graph segments comprising simple compute farms. Since no state needs to be reconstructed in case of failures, the duplicate communications are avoided.

The flow graph provides information about the runtime execution patterns of applications, allowing the framework to transparently select the appropriate recovery mechanism for the graph segments. For compute bound applications, the fault tolerance overheads during normal program execution remain low thanks to the DPS asynchronous communications that occur in parallel with computations.

The general-purpose fault tolerance mechanism allows computation to continue as long as for each thread within every thread collection either the active thread or its backup thread remains valid. The stateless recovery mechanism requires that at least one thread remains valid within every stateless thread collection, and that the threads hosting the surrounding split-merge pair are recoverable with the general purpose recovery mechanism.

Since malleability has many common requirements with fault tolerance, the fault tolerance mechanism is built on the malleability support present in DPS. The checkpointing mechanism is very similar to the thread displacement provided as part of malleability, and both mechanisms also need to be able to restart suspended operations.

From the developer's point of view, the modifications required in applications in order to support fault tolerance are minor, since most of the information required for the fault tolerance mechanism is already present within the flow graph and thread collection descriptions.

Chapter 7

Conclusion

The DPS framework provides a complete parallel programming environment for creating efficient parallel applications that can run on heterogeneous clusters. Additional features such as the support for fault tolerance, malleable applications and parallel components make DPS an interesting basis both for developing new parallel applications and for future research.

7.1 DPS key features and futures

The preceding chapters presented the DPS framework, illustrating the features made available to application developers, and how these features are implemented within the library. Let us highlight a few key features of the DPS library that have been addressed throughout this thesis, and give hints for future research.

7.1.1 Flow graphs

Flow graphs are the most noticeable feature of DPS. These provide a simple, high-level model for describing parallel applications, and can easily be represented visually. Flow graphs are built by combining user-defined operations deriving from simple primitive model elements (i.e. split, merge, process and stream). The high level flow graph representation, together with the DPS runtime, enables implicit pipelining of operations and automatic overlapping of communications and computations. With the stream operation concept, DPS expands the ability to construct very long pipelines in flow graphs. Stream operations provide a finer grain of user-controlled synchronization than the blocking split-merge constructs.

The flow graph provides a detailed description of application execution and data flow. This knowledge can be used in the framework to support additional features such as nearly transparent fault-tolerance and malleability. The framework knows when it is safe to interrupt application execution in order to move execution threads from one node to another or to take checkpoints. The flow graph is also used to deduce correct reconstruction sequences when rebuilding a state after a node failure.

The very flexible approach offered by the DPS flow graph construction provides the base for further features such as runtime integration of parallel components and recursive invocations of flow graph parts.

Future perspectives for working with flow graphs within DPS include the possibility to use a graphical user interface to design the parallel structure of an application and the data objects, and using a code generator for creating the corresponding code using the DPS library. Such an interface would make it even simpler to experiment with various parallel application structures.

7.1.2 Dynamic mapping

The dynamic mapping of threads to nodes is one of the major dynamic features of DPS. After an application has created its thread collections and populated these collections with an initial set of threads, the application can at any time alter the thread collection's contents. For simple compute farm applications, for instance, the number of threads in the computation thread collection can be changed at will in order to adjust it at runtime to the currently available number of computation nodes or to the computation power requirements of the application. Resizing thread collections for this type of application is trivial, since the threads do not store any local information.

For applications that store a distributed set of data within their thread collections, dynamic resource allocation can be achieved by using techniques such as folding or dynamic data redistribution. Folding keeps the total number of threads within the application constant but distributes threads on a varying number of nodes by moving threads from one node to another. Dynamic data redistribution alters the number of threads in the thread collection and redistributes the distributed data structure on the new set of threads. All of these approaches have been shown to work with DPS, but their impact on the performance of real applications running on dynamically evolving cluster configurations is a field for future research.

The flexible mapping of threads to nodes also serves to implement fault tolerance. Fault-tolerance can be considered as a form of involuntary thread collection alteration, where a node failure causes threads to be removed from a thread collection, or threads to move from a failed node to a backup node. The basic dynamic thread mapping mechanism is used in conjunction with a reconstruction mechanism for rebuilding on its backup thread the lost state of a failed thread. Most basic mechanisms used in fault tolerance are slightly modified versions of those used in malleability. For example, checkpointing is a variation of thread displacement, where, after displacement, the original thread is not destroyed and continues running.

7.1.3 Asynchronicity

The execution model for parallel schedules is data-driven, since the data objects circulating within the flow graph initiate the execution of operations. Operations are executed within threads, which run independently of one another. Sending and receiving data objects over the network interface is performed in separate threads, asynchronously from the threads that execute operations.

Asynchronous execution is the major driving factor in design decisions in the DPS framework. Everything the framework does, whether it is executing operations, sending data over the network, checkpointing threads, or recovering after a failure, is designed to be performed fully asynchronously. This design leads to a very high number of threads, since most tasks performed in the framework are executed within their own threads. In addition, DPS uses fine-grained locking on all its internal structures in order to ensure that ongoing tasks do not block each other for significant durations.

Using large numbers of threads puts a heavy responsibility on the operating system, which needs to schedule all these tasks, and also track the synchronization primitives. In order to minimize thread-switching overheads, DPS keeps the number of active threads to a minimum by ensuring that all inactive threads are blocking on a synchronization primitive, and that no threads perform any active polling. In order to reduce the overall number of threads, only common tasks have their own threads, and less common tasks are executed on threads borrowed from a thread pool.

7.1.4 Expandability

The DPS library was designed to be highly modular in order to simplify portability and future expansion. The most important modular components are the network layer and the connectors. Adding alternate versions of these components can allow DPS to be retargeted for networking systems other than TCP/IP.

Another objective for the replaceable components is to enable virtualization. Current work in progress implements a virtual DPS environment, where the network layer is replaced by a network simulator, and the connector simply creates new DPS controllers within the same process, as illustrated in Figure 105. This approach allows complete parallel DPS applications to be run within the context of a single process, while correctly simulating all delays and overheads that would be caused by network communications [Schaeli06]. Such a simulation environment has applications beyond simple execution time prediction: it could for example also be used to validate flow graphs in various execution scenarios or to predict system and network load for a parallel application scheduler.

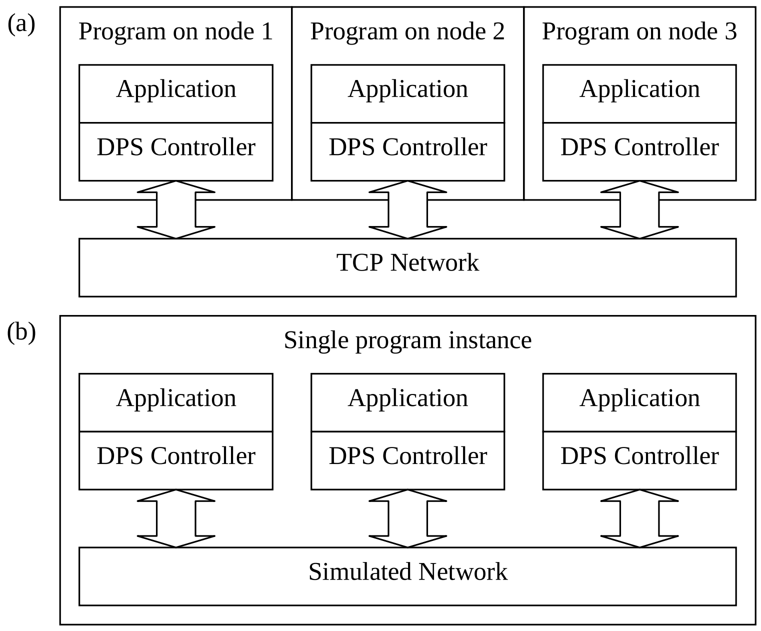


Figure 105. (a) Conventional execution of a parallel program on multiple nodes and (b) simulated execution environment within a single program instance.

7.1.5 Performance

Performance is critical in parallel applications. Therefore it is important to ensure that the framework used to develop the parallel application does not prevent the application from reaching optimal performance. In order to meet this objective, DPS is implemented in C++. This programming language is currently one of the languages of choice when performance is critical, and highly efficient compilers are available for many platforms. Also, the DPS library itself has been carefully designed in order to minimize internal overheads.

There are many overheads induced by the use of a high level library, of which the most important are the data object serialization and deserialization mechanisms, as well as the inter-operation latency. The data object serialization and deserialization has been implemented with a complex hierarchy of template classes that collapses to very simple, efficient code for handling these tasks without significant programmer intervention.

The fully asynchronous execution model provided by DPS also ensures that features such as checkpointing do not cause major performance losses, since there is no requirement to stop multiple threads simultaneously or to perform synchronous data transfers within our fault tolerance model.

7.1.6 Multi-application environments

The DPS framework provides many interesting features for multi-application environments, in particular parallel components and malleable applications. Parallel components are implemented by dynamically importing at runtime flow graph segments from one application into flow graphs defined in another application. While the feasibility of parallel components has been shown in the current implementation of the DPS framework, this feature has not yet been tested within real applications. The real-world implications of sharing parallel components with multiple client applications, in particular eventual synchronization problems, are a topic for future research.

Many parallel applications also do not need the same amount of resources over the complete duration of their execution. A typical example are applications that follow a dependency tree from the leaves to the root as illustrated in Figure 106: the application can easily be executed in parallel while processing the leaves, but the final parts can only use a few processing nodes efficiently. When multiple such applications share a single cluster, it is useful to reallocate the processing nodes to other applications when such applications are in their final processing stages.

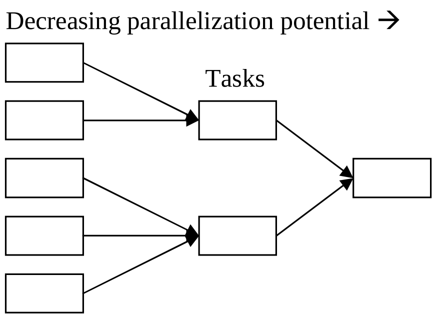


Figure 106. Decreasing potential for parallel execution in a task tree

Current ongoing research involving DPS focuses on the modeling and simulation of multiple applications running simultaneously on a shared cluster in order to optimize their aggregate efficiency.

7.2 Applications

The DPS library is a piece of middleware – it is nothing without applications that use it. Therefore, ongoing work also focuses on developing parallel applications using the framework. In particular, a recent master thesis worked on the parallelization of the popular T-Coffee package for computing multiple sequence alignments of nucleotides or amino acids [Notredame00].

The DPS library is also regularly used for teaching parallel programming at EPFL. The simple visual model of parallel programming provided by the flow graphs and thread collections makes DPS into an ideal tool for reasoning about parallel program structures.

7.3 Future perspectives

The DPS library was developed in order to provide a user-friendly approach to the development of parallel applications on clusters of commodity workstations. The fundamental assumptions for the implementation were a system with distributed memory interconnected with a commodity network running standard protocols. However, the dynamic parallel schedules programming model, with its flow graph based application descriptions, would be suitable for use on many other parallel platforms. Parallel schedules provide the developer with a simple model for constructing parallel applications comprising distributed data structures. The current implementation uses a strong object-oriented approach implemented in the C++ programming language. The current implementation requires frequent data object allocation and release, making it unsuitable for very fine-grained applications. Lighter approaches with improved memory management and better compiler support for reflection and serialization could provide an interesting basis for general-purpose programming on current and future commodity parallel systems. Implementing such a lighter approach would however require the development of a specialized compiler, in order to be able to combine the desired language traits: the performance of C/C++, the ease of object management of Java or C#, and a language facilitating the expression of dynamic parallel schedules.

Compared with classical message-passing or shared memory approaches, parallel schedules represent an alternative programming model that reduces the amount of problems that a programmer is confronted with, such as deadlocks and race conditions. Parallel schedules are well suited for large scale systems, since the model does not require any complex coherency protocols such as those present in distributed shared memory systems.

We hope that the current implementation of dynamic parallel schedules will serve as a demonstration of the potential of the programming model, both as a means for reasoning about parallel programs, and as a basis for implementing flexible, fault tolerant parallel applications.

Appendix A

CAP: A First Generation Parallel Schedule Framework

CAP is the first generation implementation of the parallel schedules concept. This appendix gives a brief overview of how CAP implements the parallel schedules concept and the syntax that is used. The results of this overview are summarized in section 3.7.

A.1 Introduction

The first generation parallel schedule framework CAP was originally created in 1998 by Benoit Gennart at EPFL. CAP implements parallel schedules by providing extensions to the standard C++ programming language. Additional language constructs and keywords allow the definition of flow graphs and thread collections directly within the source code of the application. A special preprocessor converts the special CAP constructs to standard C++ code, which is finally compiled with a regular C++ compiler.

CAP does not implement all the features of parallel schedules presented in Chapter 2, since that description is based on the current implementation DPS. Some of these limitations are highlighted in the present chapter.

Let us show how a simple parallel schedule is implemented with CAP and use it as a basis for discussing the limitations of this first generation parallel schedule framework. This discussion points to the new syntax introduced by DPS. A complete detailed description of the CAP framework is available in the CAP Reference Manual [Gennart98].

A.2 Flow graph description

The parallel schedule we consider here is a simple compute farm with two pipelined processing stages. The corresponding flow graph is illustrated in Figure 107.

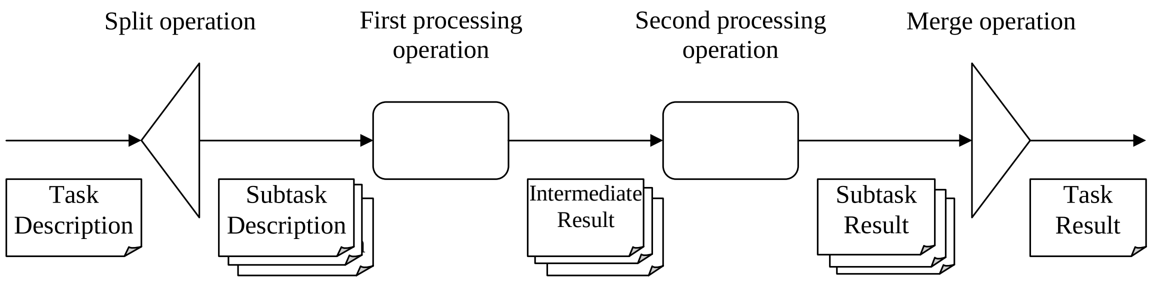


Figure 107. Simple compute farm parallel schedule with two pipelined processing stages

In the CAP framework, flow graphs are not specified directly as a graph, but by assembling predefined constructs. This flow graph requires the use of two constructs: a *pipeline* construct and an *indexed parallel* construct. The pipeline construct groups the two processing operations into a single pipeline, and the indexed parallel construct provides the surrounding split and merge operations. All CAP constructs result in a set of flow graph elements that are equivalent to a single leaf operation. The following source code shows the declaration of this flow graph in CAP.

```
// The whole flow graph
operation ExampleFlowgraph
  in TaskDescription *pIn
  out TaskResult *pOut
{
  // Indexed parallel construct
  indexed
    (int index = 0; index < 10; ++index)
  parallel
    (splitFunction, mergeFunction,
     masterThread, remote TaskResult outToken(thisTokenP))
  (
    // Pipeline of ProcessData1 and ProcessData2
    processingThreads1[thisTokenP->threadIndex].ProcessData1
    >->
    processingThreads2[thisTokenP->threadIndex].ProcessData2
  );
}
```

The declaration of these constructs references some external functions that provide the internal functionality of the operations. In particular, the indexed parallel construct references a split function *splitFunction* and a merge function *mergeFunction*. These are standard C functions that contain the code performed in the split operation and the merge operation.

```
void splitFunction(TaskDescription *pIn, SubtaskDescription *&pOut,
                   int index)
{
  // C++ statements...
  pOut = new SubtaskDescription1();
  // C++ statements to fill the pOut data object
}

void mergeFunction(TaskResult *pOut, SubtaskResult *pIn, int index)
{
  // C++ statements to store results coming from pIn into pOut
}
```

The split function is called by the CAP framework for every iteration of the loop specified in the indexed parallel construct. The split function is expected to return one new output data object on every invocation. Similarly, the merge function is called once for every input data object received by the merge operation. The merge operation should store the part of the total result present in its *SubtaskResult* input data object in the *TaskResult* output data object.

The CAP syntax, beyond being unfamiliar to the developer due to its language extensions, also has the disadvantage of spreading source code that technically belongs to the same operation into multiple places. For instance, the DPS version of this same split operation is more intuitive:

```
void splitOperation(TaskDescription *pIn)
{
  for(int index = 0; index < 10; ++index)
  {
    // C++ statements...
    SubtaskDescription1 *pOut = new SubtaskDescription1();
    // C++ statements to fill the pOut data object
    postDataObject(pOut);
  }
}
```

This shorter form has the advantage of keeping the loop and the related creation of output data objects together in the familiar C++ syntax. In addition, more context can be kept from one iteration of the loop to the next, since local variables can be declared and initialized within the split function.

The same kind of considerations can be made with the merge operation as well, since it is also called repeatedly, making it necessary to store any required context within the output data object. The allocation of the merge operation's output data object is also separated from the merge operation itself in CAP, since it is placed within the indexed parallel construct (the *remote TaskResult outToken(thisTokenP)* element allocates the output data object). All of these elements could be placed in a simple *while* loop, where all source code elements related to the merge operation are kept together. The following source code shows an example of how this construct is implemented in DPS:

```
void mergeOperation(SubtaskResult *pIn)
{
  TaskResult *pOut = new TaskResult();  // Allocate output data object
  do
  {
    // C++ statements to add elements from the pIn data object
    // to the pOut data object.
  }
  while(pIn = waitForNextDataObject());  // Wait for next data object
                                         // to be merged
  postDataObject(pOut);
}
```

The expressive power of CAP is limited by the constructs it makes available to the developer. For example, split operations can only contain loops of three types: *for* (with the indexed parallel construct shown here), *while* (with the parallel while construct), and a fixed subdivision (by explicitly specifying one split function for each output data object). Since the loops are expressed using CAP constructs, developing complex nested loops is often cumbersome, and requires using workarounds in order to obtain the desired behavior, in particular in situations where *break* or *continue* would be used in standard C code.

One of the major design objectives of DPS is to enable the expression of flow graphs using only simple, standard C++. The previous analysis of the CAP flow graph analysis has shown that the following aspects may be improved:

- Maintain all operation declarations as intuitive and readable as possible, by ensuring that all directly related source code elements remain close together. It is in particular desirable to keep the loop constructs together with the contents of the loop.
- Use standard C++ loop constructs such as *for* and *while* in order to simplify the expression of split and merge operations, and also provide maximum flexibility for the developer. Complex nested loops are simple to write.
- Keeping all elements together allows the developer to create a useful local context within split and merge operations by using local variables.

The constructs used by CAP have the same external behavior: they generate one output data object for each input data object. While this characteristic has the advantage of making the composing of constructs very simple, since any construct can be used anywhere, this limitation prevents CAP from providing constructs such as the DPS stream operation, which can receive any number of input data objects and generate any number of output data objects. The stream operation is essential for the construction of long, complex pipelines. The stream operation is a new concept that has been made possible by the more flexible design approach of DPS.

A.3 Thread collections

CAP uses a simple static model for creating thread collections. All threads are directly declared in the application's source code. The mapping of threads to processing nodes is read from a configuration file at application startup. Neither the number of threads nor the mapping of threads to processing nodes can be altered after application startup, leading to a static execution environment.

Such a static environment has a limited functionality, in particular in the presence of unreliable systems such as clusters of workstations. If one node fails during program execution, it is necessary to restart the whole application from the beginning. Also, tasks with highly varying loads such as application servers can benefit from the ability to adjust the number of processing nodes they are currently using at runtime. These aspects are addressed by DPS by introducing a dynamic thread allocation and mapping scheme.

A.4 Routing

The selection of the threads on which execution of the various operations occurs in CAP uses various mechanisms. In the previously shown flow graph definition, three of these mechanisms appear. For the leaf operation, selection is performed by prefixing the leaf operation in the flow graph declaration with the thread name and index:

```
processingThreads1[thisTokenP->threadIndex].ProcessData1
```

The *thisTokenP* variable is a pointer to the data object that is being routed to the *ProcessData1* leaf operation, and the thread index is given by the *threadIndex* member variable of the data object. The thread collection in this example is *processingThreads1*. The developer needs to provide fields in the data objects for routing information.

The thread on which the merge operation is executed is selected in the *indexed parallel* construct:

```
parallel
  (splitFunction, mergeFunction,
   masterThread, remote TaskResult outToken(thisTokenP))
```

The selected thread is *masterThread*, with no index specified since this collection contains only a single thread. The *remote* keyword indicates that the output data object of the merge operation should also be created in the context of this thread. There is no means to specify where the split operation will be executed; it is simply executed wherever the preceding operation on the flow graph took place.

As with the specification of flow graphs, the routing in CAP is not very homogeneous. Creating a single, unified approach to the declaration of routing functions is another important design aspect of the DPS approach.

A.5 Conclusion

This short overview of the CAP implementation of parallel schedules illustrates some of the more striking differences in the approach used for constructing flow graphs and thread collections. CAP provides a construct-based approach to the construction of flow graphs. Whereas this approach allows for simple application design by limiting the choices offered to the developer, it sacrifices the flexibility found in the general parallel schedule model. In particular, developers have little control over split operations and merge operations, since most of their functionality is controlled by the construct that is used. The parallel

CAP constructs provide a 'fill out the gaps' approach where some elements are fixed by the construct, in particular the looping model within split operations, and some other elements are filled out in various places in the application source code.

The language extensions used in CAP require the use of a specialized preprocessor in order to parse CAP source code. The use of language extensions puts an additional load on application developers, who need to familiarize themselves with these extensions, as well as on the maintainer of the preprocessor, since parsing and transforming C++ source code with extensions is by no means a trivial task.

CAP also provides a static execution model, similar to what is used in MPI. The application starts with a given mapping on a fixed number of machines, and this mapping is never modified at runtime. If a single participating machine dies, the whole application dies with it.

The introduction of the first implementation of parallel schedules by CAP was an important step towards the creation of a high-level framework for the development of pipelined parallel applications [Messerli99]. The improvements brought by DPS now allow parallel schedules to be use in a wide range of contexts such as fault-tolerant parallel applications, parallel components, and parallel applications with dynamically varying allocation of resources.

Appendix B

DPS Tools

The DPS framework enables the development of parallel applications and provides a runtime system for executing these applications. However, the DPS framework would not be complete without a set of accompanying tools for monitoring application behavior, cluster usage, and debugging.

B.1 Introduction

A consistent set of tools for monitoring parallel applications is an invaluable resource for evaluating and optimizing application performance. DPS provides a set of tools for monitoring clusters of workstations and individual applications, both in real time and offline. The monitoring tools provide a graphical user interface. In order to ensure that the tools run consistently on all the platforms supported by DPS, they are written in Java. Two tools were developed: *DPS Control Center*, which performs all the real time monitoring tasks on applications and clusters, and *DPS Trace Analyzer*, which performs offline application performance analysis.

B.2 DPS Control Center

The DPS Control Center application performs two major real time tasks: cluster management and application execution analysis. The cluster management component can be used for monitoring any number of workstations, and for displaying their current status in its user interface. A typical view for a room of Sun workstations is shown in Figure 108. For each workstation, the following parameters are displayed:

- CPU usage, both user and kernel times
- Network interface usage, inbound and outbound traffic
- Memory usage, physical and swap file
- Number of active processes and users

The monitoring information is obtained from the DPS kernels running on the workstations. The monitoring code is derived from the open-source package *GKrellM* [GKrellM]. This package provides a client-server based system monitoring tool with a customizable user interface. The user interface is tailored for monitoring single machines, and therefore unsuitable for monitoring large clusters. We have therefore redeveloped a new user interface within the DPS Control Center application. The server component of GKrellM is cross platform, providing monitoring for Windows, Linux, Solaris and many other systems. It was therefore well suited for integration into the DPS kernel. The server component had some external dependencies, in particular various GTK libraries, which were not desirable within the context of DPS. These dependencies were therefore removed and replaced by simpler substitutes. The network protocol used by GKrellM was left untouched, allowing full interoperability of DPS Control Center, DPS kernels, and original GKrellM clients and servers.

Based on the monitoring data, the DPS Control Center can automatically generate lists of machines that satisfy a certain set of criteria. A typical use is to select machines that have no users logged in interactively and that have an average CPU load below 10%. These lists are subsequently used with the pattern mapper presented in section 3.5.4 in order to generate mapping strings for the thread collections of the parallel applications running on the monitored cluster.

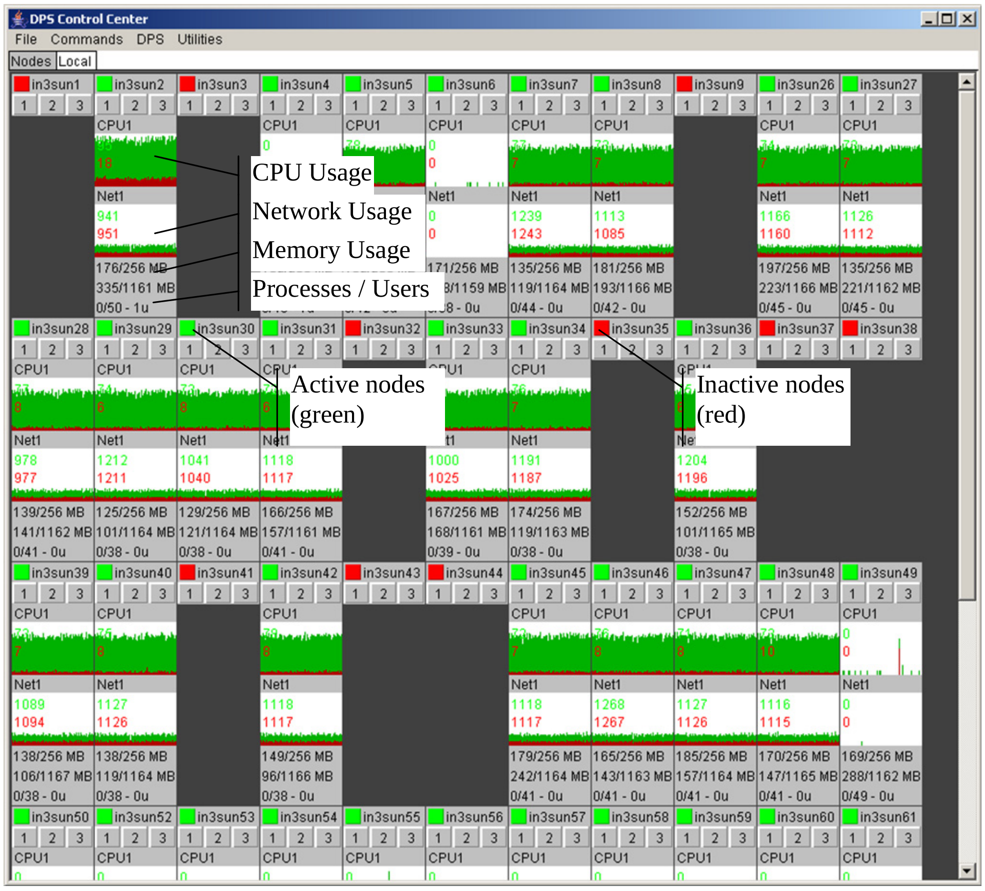


Figure 108. DPS Control monitoring of a room of Sun workstations, currently running a parallel application on 20 nodes

B.2.1. Application startup

The DPS Control Center can also be used for starting and stopping applications running on the cluster. For these tasks, the control center uses the standard *ssh* secure shell. Applications can be started both on individual nodes and on all nodes simultaneously. This is in particular useful for starting the DPS kernel on a large number of machines.

The DPS control center also has the ability to execute scripts involving parallel applications. A simple script for running a hypothetical parallel application *parapp* seven times is shown below:

```
1:0:parapp -pat  4x1+0  -port 6005 -map nodes.txt
1:4:parapp -pat  4x1+4  -port 6005 -map nodes.txt
1:8:parapp -pat  4x1+8  -port 6005 -map nodes.txt
1:12:parapp -pat 4x1+12 -port 6005 -map nodes.txt
@1
1:0:parapp -pat  8x1+0  -port 6005 -map nodes.txt
1:8:parapp -pat  8x1+8  -port 6005 -map nodes.txt
@1
s:0:parapp -pat 16x1+0  -port 6005 -map nodes.txt
```

Each line of the script is composed of three elements separated by colons. The first element is a synchronization identifier, which is used to control the execution of the script. The second element is the index of the node (0-based) on which the application should be executed. The third element is the command line to be executed. The applications are executed in the order indicated in the script, without waiting for the application to terminate before moving on to the next line. The lines starting with the @ character indicate that the system should wait until all tasks with the same synchronization identifier appearing after the @ have terminated. The above script first runs four instances of the application on 4 nodes each, followed by two instances on 8 nodes each, and finally performs a single run on 16 nodes. The nodes used for executing the application are automatically selected from the set of idle nodes currently running the DPS kernel, and are output to a *nodes.txt* file appropriate for use with the pattern mapper (see section 3.5.4).

B.2.2. Application monitoring

The other main feature of the DPS Control Center is the monitoring of individual DPS applications. Figure 109 shows a typical run of the game of life application presented in section 4.2, monitored by the DPS Control center. The application view is highly detailed, displaying a complete view of the parallel schedule as it executes. For each operation that was started, its name, thread collection, and thread index is displayed. The queue of data objects waiting within the executors for processing is displayed along the operations that perform the processing. Finally, for split/merge operation pairs, the current state of the feedback mechanism for the merge operation termination and flow control (see section 5.3.2) is also displayed.

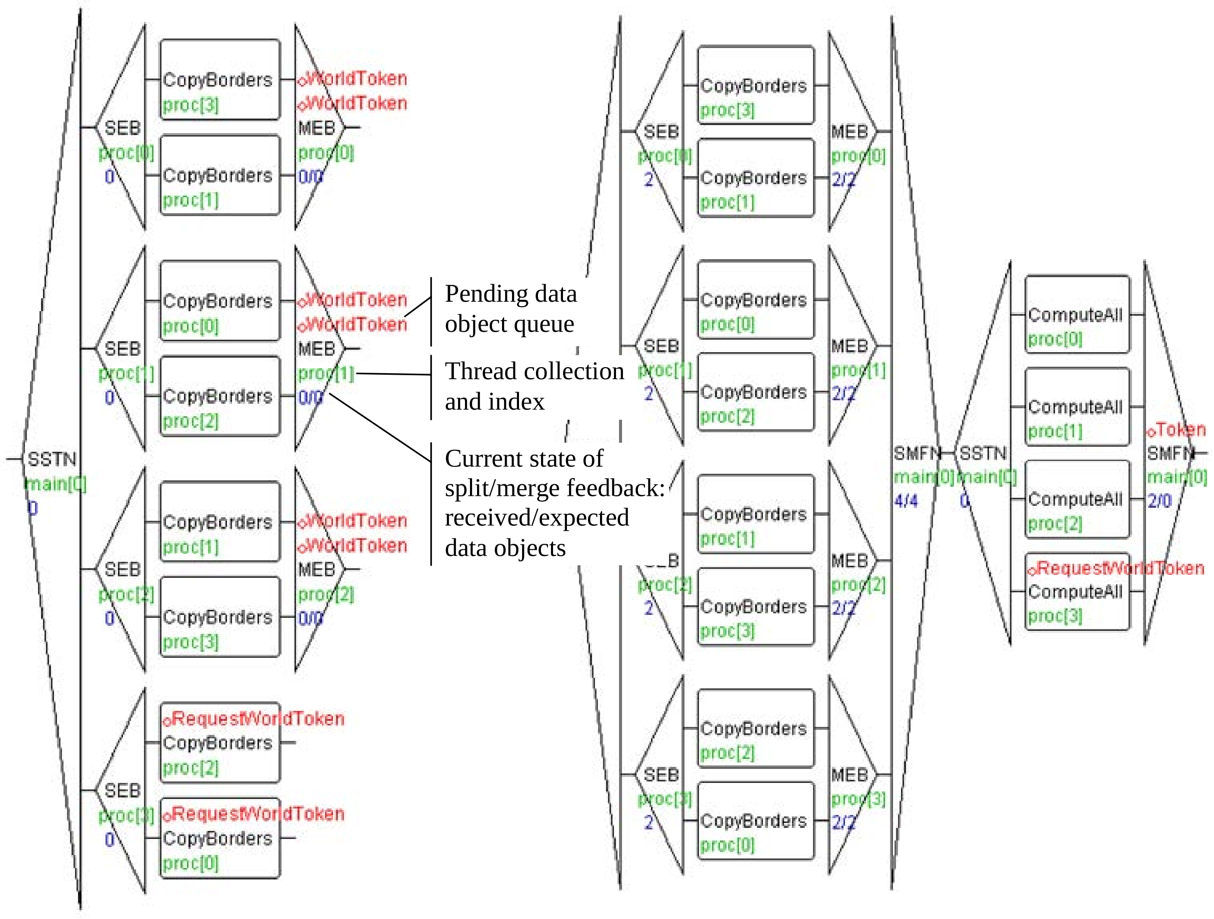

SSTN
main[0]
0
SEB
proc[0]
CopyBorders
proc[3]
proc[1]
MEB
WorldToken
Pending data object queue
Thread collection and index
Current state of split/merge feedback: received/expected data objects
RequestWorldToken
ComputeAll
SMFN
main[0]
4/4
2/0
Token
2/2

Figure 109. DPS Control Center monitoring of the game of life during border exchange (left) and during end of iteration (right)

The flow graph execution monitoring feature requires the application to interact with the DPS Control Center for every action performed within the DPS framework: creating operations, posting of data objects, split and merge notifications, and many others. This interaction is based on a simple network protocol where the running application sends a notification to the DPS Control Center before performing any action, and waits for an acknowledgement before proceeding. The application sends all potentially useful information to the DPS Control Center at a very high level of detail, including the contents of the transmitted data objects. Due to the large overheads involved, the application runs slowly when application monitoring is active. This tool is very useful for debugging, since it gives a precise idea of the applications behavior, providing insight about where operations are executed and about the number of data objects in circulation at a given time.

B.3 DPS Trace Analyzer

The other tool provided with the DPS framework is an offline trace analyzer. A running DPS application can dump all its activities and the corresponding timestamps to a log file. These log files can be viewed graphically with the trace analyzer. As an illustration of the output of the trace analyzer, we show several traces extracted from the LU decomposition application. The LU decomposition uses the parallel schedule presented in section 4.4.2, with pipelining disabled. The application is running on 8 nodes, with 12 matrix block storage threads and 8 multiplications threads. The flow graph for the LU decomposition is illustrated in Figure 110.

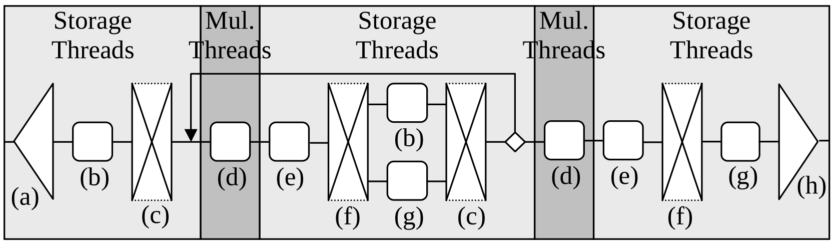


Figure 110. Flow graph for LU factorization. The multiplication operation (d) is executed within its own thread collection.

The successive operations involved in the LU decomposition are summarized below:

a) Perform Gaussian elimination for top left block A_{11}.

b) Solve in parallel the triangular system in order to compute U_{12} for all other column blocks.

c) Stream out multiplication requests as each column completes the triangular system solve.

d) Parallel block-based matrix multiplication to compute $L_{21} \cdot U_{12}$.

e) Subtract the result of the multiplication from A_{22} in parallel.

f) As soon as the multiplication for the first column of blocks is complete, perform the next level Gaussian elimination, and stream out triangular system solve requests as soon as the multiplications for the other column blocks complete.

g) Perform row flipping on previous column blocks according to permutation matrix P.

h) Wait for the final row exchanges to complete before leaving applications.

The stream operations (c) and (f) are used in order to ensure optimal pipelining of the successive steps of the block-based LU decomposition. A trace of the execution of the pipelined LU decomposition is shown in Figure 111. The overlapping of the successive iterations of the decomposition (stream operations (c) and (f)) is clearly visible, with up to three iterations running in parallel.

The trace shown in Figure 112 shows the successive initial stages of the LU decomposition with the pipelining disabled. For this execution, the stream operations (c) and (f) have been modified in order to ensure that the stream operations only start generating output data objects once all their expected input data objects have been received. The modified stream operations achieve the same functionality as a merge operation followed by a split operation.

Within its user interface, the trace analyzer provides additional information on operations, such as the operation type and the operation's starting and ending times. This information is accessed by simply holding the mouse cursor over any operation.

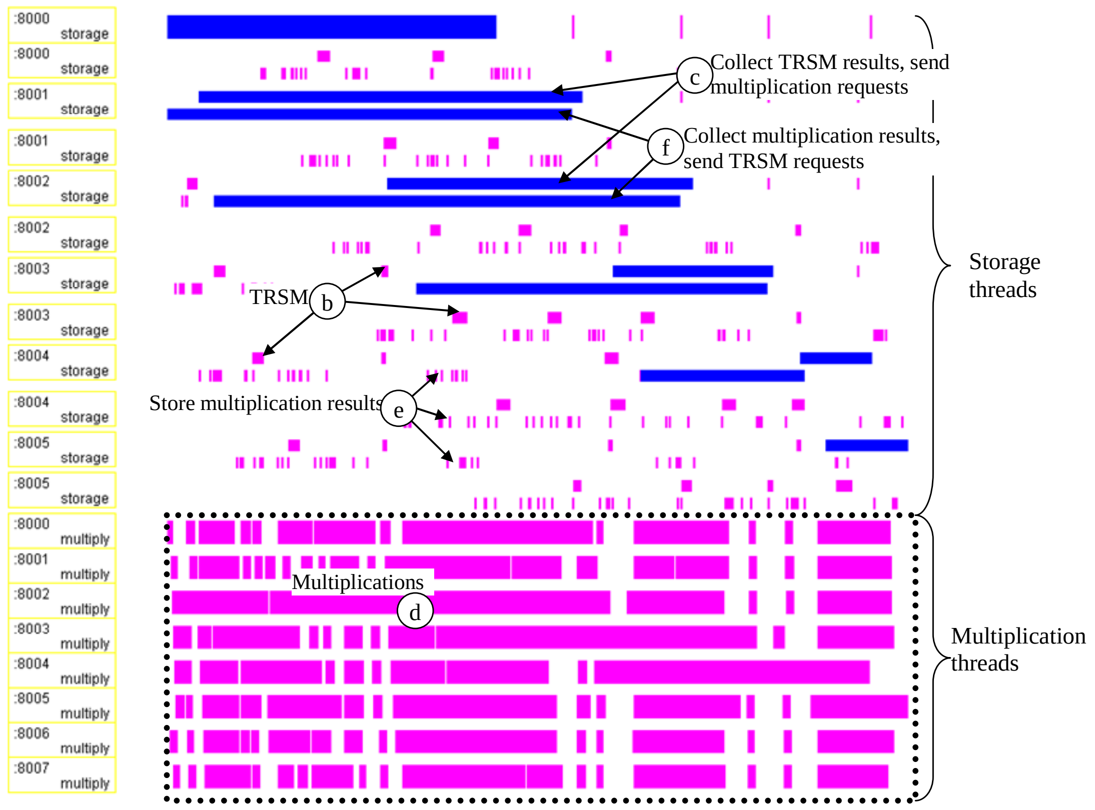


Figure 111. Output of the trace analyzer for part of the LU decomposition with pipelining enabled

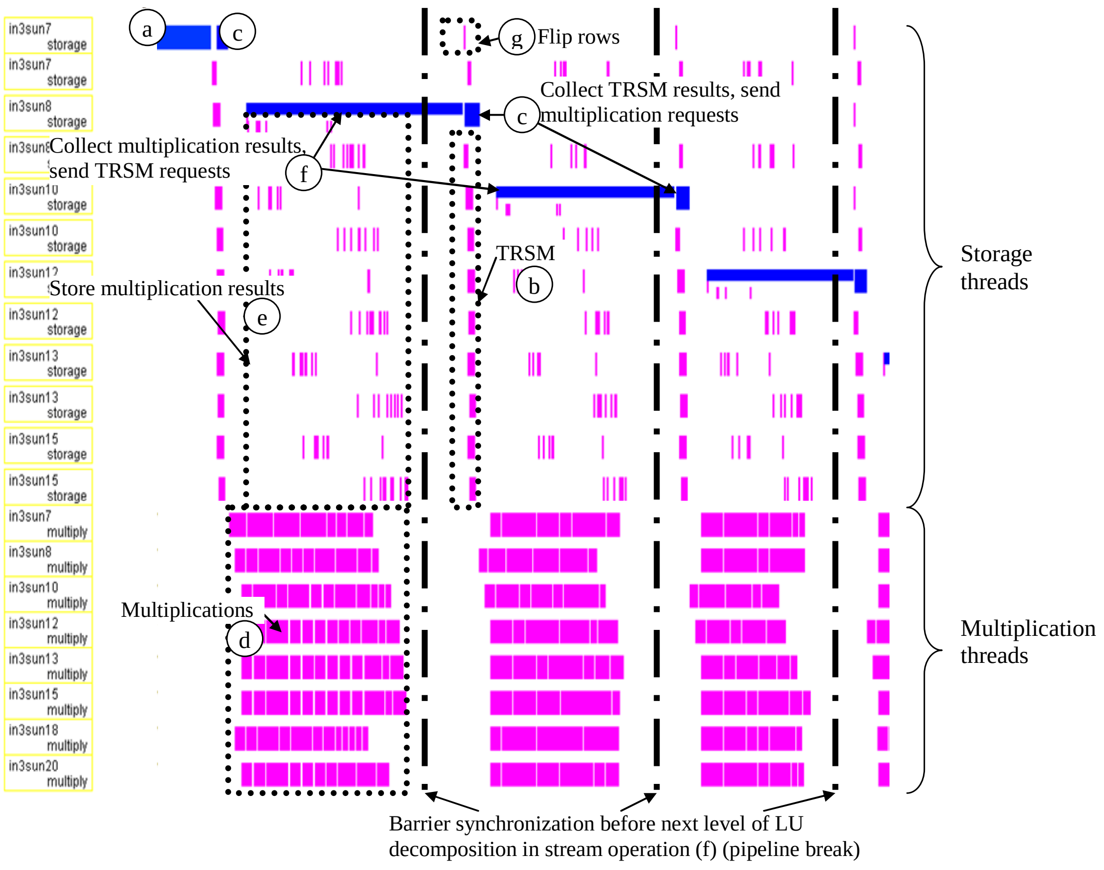


Figure 112. Output of the trace analyzer for part of the LU decomposition with pipelining disabled

The previous example shows how the trace analyzer can detect inefficient resource usage through the lack of pipelining within a parallel application. The view of operations provided in the trace analyzer is also particularly useful for detecting other situations, such as bottlenecks or load imbalances within parallel applications. Trace analysis is an important tool enabling developers to optimize an application's performance.

Beyond displaying operations, the trace analyzer offers various views of the data objects circulating within the parallel application, allowing specific sequences of events to be precisely examined. It is in particular possible to examine the time that data objects spend in the executor queues, which is useful for determining the optimal flow control limits for split-merge pairs.

B.4 Conclusion

The tools provided in this appendix enable users to manage DPS applications running on their clusters and to gain a detailed view of the runtime behavior of their parallel applications. The DPS control center provides a simple user interface for monitoring large numbers of machines and generating mappings for deploying parallel applications on idle nodes. The DPS trace analyzer provides a detailed offline analysis of parallel schedule execution timelines.

Appendix C

C++ Internals

The presentation of the internals of DPS in Chapter 5 avoided most of the complex C++ code present in the DPS library by illustrating only the external interfaces of most of the presented mechanisms. This Appendix provides an overview of the C++ programming techniques used internally in DPS, and is intended for readers having an advanced knowledge of C++.

C.1 Introduction

In this Appendix, we present in more depth the internals of the DPS library, beyond the explanations presented in Chapter 5. We provide to the interested reader additional information on topics related to type handling and data serialization/deserialization. In order to understand the source code samples shown here, advanced knowledge of C++ is required, in particular partial template specialization [ISO05].

There are still many more classes within the DPS framework that are not presented here. The interested reader may examine the library's reference documentation or the source code.

C.2 Type naming

We have seen in section 5.2.2 that DPS needs to be able to determine the type of objects at runtime in order to allow the transmission of the type information from one node to another. The C++ language provides a standard features for obtaining type information at runtime, called RTTI (Run-Time Type Information). However, RTTI does not provide consistent type information across compilers and platforms. Since DPS needs to transfer objects between heterogeneous platforms, we need to provide our own mechanism for representing type names.

The simplest approach to providing type information from within objects is to add a single static member function *getTypeName* within the type. For most user-defined types used within the DPS library, this method is provided by the *IDENTIFY* or *CLASSDEF* macros as shown below:

```
// CLASSDEF(Foo) or IDENTIFY(Foo) (partial macro expansion)
static const char *getTypeName() { return "Foo"; }
```

This member function can only easily be added to user-defined types. Therefore, this approach will not work with types provided by the system libraries, such as the STL classes, or even with simple built-in types such as *int* or *char*. In order to handle these cases, we use a template class *TypeHandler*, which provides a static method *name* that returns the string equivalent of the type passed as a template parameter. The following source lines are the generic declaration of *TypeHandler*:

```
// Generic form
template<typename x> struct TypeHandler
{
  static inline const char *name() { return x::getTypeName();}
};
```

This generic form assumes the presence of a static member function *getTypeName* within the type that is used as template parameter. For other types, such as the basic C++ types, DPS provides specialized forms of *TypeHandler*. The following example shows the declaration for *char*:

```
// Specialized form of TypeHandler for char
template<> struct TypeHandler<char>
{
  static inline const char *name() { return "char"; }
};
```

For other common types which are not user-provided, such as the types provided by the Standard Template Library (STL), DPS also provide explicit specializations:

```
template<> struct TypeHandler<std::string>
{
  static inline const char *name() { return "std::string"; }
};
```

Partial specializations of the type handler class are also implemented for all the STL container classes:

```
template<typename x> struct TypeHandler <std::vector<x> >
{
  static inline const char *name()
  {
    // If x is int, the following macro returns "std::vector<int >"
    RETSTRCAT3("std::vector<",TypeHandler<x>::name()," >");
  }
};
```

The *RETSTRCATx* macro returns a string containing the concatenation of the provided arguments. The string is generated with a singleton pattern, ensuring that it is allocated and created only once.

Another case requiring partial specialization is the naming of array types. Here again, *RETSTRCATx* is used to generate a singleton string, and a helper class *IntToStringConverter* is used to convert the array size to a string:

```
template<int size> struct TypeHandler<x [size]>
{
  static inline const char *name()
  {
    // Use TypeHandler to retrieve the name of the base type, and
    // append the array dimensions after it. This will work recursively
    // for multi-dimensional arrays.
    RETSTRCAT4(TypeHandler<x>::name(),
      "[", IntToStringConverter(size).value, "]");
  }
};
```

User-defined template classes also require particular attention, since they will also have to return a unique name for every possible template argument. This cannot be done by using the simple *CLASSDEF* macro presented before, since it does not know the template arguments. Therefore, for template classes, the *CLASSDEF* macro is replaced by a *TEMPLATEDEFx* macro that takes one additional parameter for each template argument:

```
template<typename myType> class FooTemplate
{
  TEMPLATEDEF1(FooTemplate,myType)
  MEMBERS
    ITEM(myType,myMember)
  CLASSEND;
};
```

The *TEMPLATEDEFx* macro generates a proper *getTypeName* function including all template parameter type names. For example, in the above case, the returned string is "FooTemplate<int >" if *myType* is *int*.

Beyond its use for retrieving unique, cross-platform type names, *TypeHandler* is also used for handling serialization and deserialization of the data types. This usage is presented in section C.6.

C.3 Class factory

When DPS receives a type name returned by the mechanism shown in the previous section, it needs to be able to create an instance of that type. Creating objects from a type name is achieved by using a class factory. A traditional implementation of class factories in C++ uses global variables in order to register the types in the class factory. The following lines of source code show a typical implementation (the code itself has been omitted for brevity, only the general structure is shown):

```
template <typename t> class RegisterClass
{
  public:
    RegisterClass() { /* Register class in class factory here */ }
};

class MyClass { /* Class declaration */ };
// Register class in class factory
RegisterClass<MyClass> registerMyClass;
```

Since *registerMyClass* is a global variable, this code will cause the constructor to *RegisterClass* to be called at program startup, thereby registering the class with the class factory. The class factory can then internally store a string representation of the type passed as a template parameter, and create objects of that type. The main problem with this approach is that the global variable needs to be declared outside of the class declaration, and therefore requires an additional declaration from the developer. The class needs one macro inside its declaration for the type naming, and another macro outside its declaration for the registration in the class factory.

In order to avoid this cumbersome declaration, DPS uses an additional indirection with two template classes in order to move the registration within the class factory into the class scope. This makes it possible to generate the registration within the *IDENTIFY* or *CLASSDEF* macros. These macros simply add an additional member function to the class as follows:

```
virtual Size getFactoryTypeIndexV() const
{
  // The template parameter x below is the type of the current class
  static dps::GenericConstructorHolder<x > gch;
  return gch.getIndex();
}
```

This function simply returns the index within the class factory of the type. The important part of this function for the registration process is that the function references a static variable of type *GenericConstructorHolder*. This type, however, cannot be used for registering the class within the class factory, since most compilers will initialize this type of local static variables only when the function is called for the first time. Therefore, we use a second type, *GenericConstructor*, to declare a static variable within *GenericConstructorHolder*. This static variable being at the class level, the compiler generates code for its initialization in a similar fashion to global variables, thereby ensuring that our class is registered at startup from within the constructor of *GenericConstructor*. The use of templates ensures that the compiler generates one instance of *GenericConstructor* for each type to be registered in the class factory.

```
template<typename x> class GenericConstructorHolder
{
private:
  // The static variable at class scope that is used to effectively
  // register class x in the class factory
  static GenericConstructor<x> item;

public:
  // Constructor
  GenericConstructorHolder() { item.name=regc::getTypeName(); }

  // Get index in Registrar
  static Size getIndex() { return item.getIndex(); }
};
```

C.4 Type lists

The type matching mechanism shown in section 5.2.5 used a template class *tv* for storing collections of types. Internally, the collection of types is implemented as a type list. A type list is similar in structure to a linked list. Each element in the type list is a C++ template class, where one of the template arguments is the next element in the list. A special type is used as list terminator. The following example shows a simple type list declaration:

```
// Declaration of a type list member
template<typename element, typename next> class TypeVector
{
public:
  typedef element t0;   // Make template arguments available
  typedef next t1;      // to external users of the type list
};

// Type list with one element, a
dps::TypeVector<a, dps::TypeVectorEnd>

// Type list with two elements, a and b
dps::TypeVector<a, dps::TypeVector<b, dps::TypeVectorEnd> >
```

Since all elements in the list are types, the list and all operations on it are handled at compilation time, inducing no overhead at runtime. The individual elements of the list are of the type *TypeVector*, and the last element is a *TypeVectorEnd*. The *TypeVector* class contains additional typedefs in order to allow navigation within the elements of the list.

The *tv* template used in DPS for type lists is simply a template wrapper around the type list model shown above in order to provide a cleaner interface for the developer. The basic form of the *tv* template is designed for holding five elements, and specialized variants are available for holding one to four elements. The following source code shows the specialized implementation for the 3-argument version of *tv*.

```
template<typename a, typename b, typename c>
  class tv<a,b,c,None,None> : public
    dps::TypeVector<a,                // First type
      dps::TypeVector<b,              // Second type
        dps::TypeVector<c,            // Third type
          dps::TypeVectorEnd> > >     // List terminator
{
  public: typedef a Type1; // Shortcut to first type
};
```

Based on these type vectors, DPS provides several type matching templates. The first of these, *MatchTypeType*, simply determines whether two types are identical or not, returning the result in an *enum*. The implementation of these templates uses simple partial template specialization.

```
// Simple type matching, generic declaration for non-matching types
template<typename t0, typename t1> struct MatchTypeType
{ enum { result=0 }; };

// Simple type matching, specialized declaration for matching types
template<typename t0> struct MatchTypeType<t0,t0>
{ enum { result=1 }; };
```

Beyond simple type matching, DPS also provides templates for matching a type within a type vector, and for finding out whether two type vectors contain at least one common element. These templates also return their result in an *enum*, and use a recursive descent through the type vector to find matching elements. The following shows the code for finding a matching type within a type vector:

```
// Find type in type vector
template<typename tv, typename ts> struct MatchTypeTypeVector
{
  // First check whether the current type matches the input type
  // The current type is available in t0
  enum { result = MatchTypeType<typename tv::t0,ts>::result |
  // and the recursively check the next type available in t1
                MatchTypeTypeVector<typename tv::t1,ts>::result };
};

// Find type in type vector (end marker)
template<typename ts> struct MatchTypeTypeVector<TypeVectorEnd,ts>
{
  enum { result=0 };
};
```

The same mechanism is used for implementing the *MatchTypeVectorTypeVector* class, which tries to find a matching type between two type vectors. In the implementation of the right shift vector used for combining *FlowgraphChain* items, the *MatchTypeVectorTypeVector* class is used to validate the sequence of elements. For this validation to work, all types actually need to be type vectors, but we do not want to force the developer to use the *tv* template in all the cases where only a single type is allowed, since this is the most common case. Therefore the operation base classes and others that require type vectors have been amended with an additional specialization that automatically creates type vectors when a single type is specified. Therefore the following operation declarations are identical:

```
class MyOperation : public dps::LeafOperation
  < dps::tv<InputType>, dps::tv<OutputType> >
{ };

class MyOperation : public dps::LeafOperation
  < InputType, OutputType > // The missing tv's are added by the library
{ };
```

The missing *tv* template classes are added by using a template class *MultiTV* that uses partial specialization to enclose its argument in a *tv* when it is not of type *tv*. All arguments passed to operations are internally wrapped with *MultiTV*, and the type lists are accessed only through the internal *TypeVector* typedef in *MultiTV*, thereby ensuring that DPS deals only with type vectors.

```
// Type vector wrapper, generic form without tv
template<typename a> struct MultiTV
{
  // create a new type vector containing one element
  typedef tv<a> TypeVector;
};

// Specialized type vector wrapper
template<typename a, typename b, typename c, typename d, typename e>
  struct MultiTV<tv<a,b,c,d,e> >
{
  // simply copy the input type vector
  typedef tv<a,b,c,d,e> TypeVector;
};
```

C.5 Data reflection

In order to successfully serialize and deserialize data, DPS needs to know the member variables contained within a class. The macros used for obtaining this information were briefly shown in section 5.5.3. We will now examine how these macros generate the required list of member variables. For each member variable, the framework needs to know its type and be able to read and write the value of the variable. In addition to the member variables, we also need to know which base classes the class derives from, in order to also allow the enumeration of all inherited member variables.

The reflection information needs to be stored and accessible without incurring any runtime overheads. Since classes are statically defined, type lists are a good choice for storing such information, since they can be entirely processed at compile time, and generate efficient code for handling their elements.

The following source code shows an example of the macros used by DPS for a simple class:

```
class Foo : public Bar
{
  CLASSDEF(Foo)
    BASECLASS(Bar)
  MEMBERS
    ITEM(int,a)
    ITEM(int,b)
  CLASSEND;
};
```

In order to generate the type lists from the macros shown above, we use *typedefs* that begin in one macro and end in the next in order to link the elements together. The following source code illustrates the expanded macros for generating the lists of base classes and members:

```
typedef LastMember                           // CLASSDEF(Foo)
prevbc_4; typedef Member<prevbc_4,Bar>       //   BASECLASS(Bar)
classBases; typedef LastMember               // MEMBERS
prev_a; int a; typedef Member<prev_a,int>    //   ITEM(int,a)
prev_b; int b; typedef Member<prev_b,int>    //   ITEM(int,b)
classMembers;                                // CLASSEND
```

With normal line breaks restored for improved readability, the generated code is:

```
typedef LastMember prevbc_4;
typedef Member<prevbc_4,Bar> classBases;
typedef LastMember prev_a;
int a;
typedef Member<prev_a,int> prev_b;
int b;
typedef Member<prev_b,int> classMembers;
```

One list is created for base classes (entry point *classBases*), and another for the members (entry point *classMembers*). The *Member* template class is used to represent the class members in the list. It contains visitor functions that are used for carrying out operations on the member items. For example, the following visitor function can be used to print the types of all members. The list is traversed by simple recursion.

```
template <typename prev, typename item> struct Member
{
  static void printTypeNames()
  {
    prev::printTypeNames();
    std::cout << TypeHandler<item>::name();
  }
};

struct LastMember
{
  static void printTypeNames() { } // Do nothing
}
```

In the above example, the *TypeHandler* template class is used to obtain a string representation of the type passed as template parameter. This class is described in section C.2. With the above example, the following call would print the type names of all members of class *Foo*:

```
Foo::classMembers::printTypeNames();
```

When used with a compiler that properly handles inline functions, the call to *printTypeNames* will be reduced to a simple list of calls to the << operator. Similarly, the names of all base classes can be printed by calling *printTypeNames* on *classBases*.

When writing objects to structured formats, such as XML, it is necessary to know the names of the members as well as their types. The strings have to be encapsulated in classes in order to be valid as template parameters. The encapsulation class can easily be created as part of the *ITEM* macro, and then used as a parameter to the *Member* template.

```
// ITEM(int,a) (partial macro expansion)
struct name_a
{
  static inline const char *name() { return "a"; }
};
```

The string encapsulation class contains a single static function that returns a constant string pointer. The function is inlined by optimizing compilers, producing no overhead. The *Member* template now has three arguments.

```
// ITEM(int,a) (partial macro expansion)
typedef Member<prev_a,int,name_a>
```

Knowing the name and type of the members of a class is not sufficient for serialization and deserialization: we also need to be able to access the values of the reflected members. Since the *Member* template class is separate from the user-defined class and pointers to class members cannot be passed as template parameters, a mechanism is required to provide access to the values of these class members. This is achieved with an additional helper class called *accessor*. This class provides static functions that will return a reference to a particular member of a class, with a reference to an instance of the class passed as input. It is generated by the *ITEM* macro as follows:

```
// ITEM(int,a) (partial macro expansion)
struct accessor_a
{
  typedef int memberType;
  // Get reference to member
  static inline int& get(containerType& instance)
  {
    return instance.a;
  }

  // Get const reference to member
  static inline const int& getConst(const containerType& instance)
  {
    return instance.a;
  }
};
```

Both static methods *get* and *getConst* take an argument of type *containerType*. The *containerType* typedef provides a means to recover the type of the current class in the other macros following *CLASSDEF*, since C++ provides no means to recover the type of the current class.

```
// CLASSDEF(Foo) (partial macro expansion)
typedef Foo containerType;
```

The *Member* template class thus requires both the *accessor* and the type of the object containing the members we wish to serialize as template arguments. The type of the member can be retrieved from the *accessor* class, and is not required as a template argument. Therefore the *Member* template used for describing an item is redefined as:

```
typedef Member<prev_a,accessor_a,containerType>
```

With the accessors, the type list composed of *Member* elements now provides a complete mechanism for reflecting and accessing the members of a class. The resulting reflection provides a simple means of obtaining the names and types of the individual members of a class independently of the compiler. The order of the reflected members is also independent of any compiler-generated initialization orders thanks to the fixed sequence of *typedefs*. The accessors provide a means to change the values of the members reliably, without needing to know their exact position in memory, making the system robust to changes in structure member alignment. These features serve as a base for building a reliable cross-platform serialization mechanism.

C.6 Serialization

We have shown how the reflection mechanism described in section C.5 can list all the members of a class and retrieve their types, names and values. The serialization mechanism is composed of two layers. First, the *TypeHandler* template class presented in section C.2 is extended to support reading and writing of the types it represents, traversing the complex types until it reaches simple types. These simple elements are passed on to streams that read the data from the appropriate source or write it to the appropriate destination. These streams provide a common interface, *IInputStream* or *IOutputStream* depending on the direction of data transfer.

In the following discussion, we consider only the writing functions. The reading functions rely on the same principles.

C.6.1. Member traversal for serialization

The *TypeHandler* is extended for performing writing operations by adding a write static inline member function. Like the previously presented *name* function used for retrieving type names, a generic form is provided that expects the presence of a *write* member function in the type. This write function is responsible for writing the object to the stream. For all simple types, specialized forms that directly call the appropriate formatting function in the stream object are implemented.

```
// Generic form
template<typename x> struct TypeHandler
{
  static inline void write(IOutputStream *s, const x& item)
  {
    item.write(s);
  }
};

// Specialized form for char
template<> struct TypeHandler<char>
{
  static inline void write(IOutputStream *s, const char& item)
  {
    s->writeInt8(item);
  }
};
```

The write member function required for user defined types is generated by the *CLASSDEF* macro. This function initiates the traversal of the list of base classes and members, calling a write visitor function in the *Member* template class:

```
// CLASSDEF(Foo) (partial macro expansion)
void write(IOutputStream* s) const
{
  classBases::write(s,*this);
  classMembers::write(s,*this);
}
```

The required visitor functions are created by the *ITEM* macros to sequentially serialize all member items. The *write* visitor function walks through all members of the class, using the *accessor* to retrieve the member values.

```
template <typename prev, typename accessor, typename container>
  struct Member
{
  static void write(IOutputStream *s, const container& instance)
  {
    prev::write(s,instance);
    typehandler::write(s,accessor::getConst(instance));
  }
}
```

The data that is retrieved by the *accessor* class is transferred to the appropriate type handler that will handle the writing of the specific data type. Since the return type of the *accessor* is not explicitly available, a function template is used to access the *TypeHandler*. This function template resides in its own namespace, and takes advantage of the implicit template parameter deduced from the function arguments in order to forward the call to the appropriate specialization of the *TypeHandler* class.

```
namespace typehandler
{
  template<typename x> void write(OutputStream *s, const x& item)
  {
    // Use the appropriate form of TypeHandler with the type
    // deduced from the function argument
    TypeHandler<x>::write(s,item);
  }
}
```

For specific types such as the STL containers, the specialized forms of the *TypeHandler* class have been extended to support serialization by using an iterator to walk through all the elements of the container. The individual elements are passed to the write function of the type handler mechanism, allowing them to be of any type that can be serialized.

```
template<typename x> struct TypeHandler<std::vector<x> >
{
  static inline void write(IOutputStream *s, const std::vector<x>& item)
  {
    typename std::vector<x>::const_iterator it;

    // Cast to UInt32 to ensure consistent size across platforms
    UInt32 size=(UInt32)item.size();

    // Write the size to the output stream
    typehandler::write(s,size);

    for(it=item.begin();it!=item.end();it++)
    {
      // Write individual items to output stream
      typehandler::write(s,*it);
    }
  }
};
```

All the functions used in the traversal phase are inlined by the compiler on optimized builds, thus yielding very simple executable code. It is usually just a list of calls to the *IOutputStream* interface for every simple element within the object. The complex call hierarchy generated by the visitor functions, accessors, and type handlers is fully collapsed.

C.6.2. Formatting of serialized output

The type handler transforms complex object serialization requests into simple type writing requests. Simple type writing requests are handled by a class derived from the abstract base class *IOutputStream*. *IOutputStream* provides pure virtual functions for writing all simple types available in C++.

```
class IOutputStream
{
public:
  virtual void writeInt8(const Int8& item) = 0;
  // ... Additional types
  virtual void writeBool(const Bool& item) = 0;

  // For high performance writes of blocks of simple types
  virtual void writeArray(const void *data,
                          const Size count, const Size size) = 0;
};
```

DPS provides an initial set of classes deriving from *IOutputStream* that handle initial formatting of the data: *BasicBinaryWriter* and *BasicTextWriter*. These classes format the data into binary or text output. Other classes are then derived from these two basic writers to interact with the final destination of the data, such as *BinaryFileWriter* or *BinarySocketWriter*. These classes deriving from *BasicBinaryWriter* are very short, since *BasicBinaryWriter* simply provides a set of memory blocks that have to be transferred to the destination. *BasicBinaryWriter* is optimized to minimize memory copies, since calls to *writeArray* will simply store the passed pointers rather than copying the data to an output buffer.

Endian flipping on binary transfers is always performed at the receiving end, by providing two readers deriving from *IInputStream*: *BasicBinaryReader* and *BasicBinaryFlippingReader*. Within DPS, endian compatibility checks are performed on a per connection basis, and the appropriate reader class is used based on the comparison result. When two machines with the same endianness communicate, no flipping is performed.

C.6.3. Pointers and polymorphism

Many complex data structures, such as linked lists or trees, are implemented by using pointers. In order to serialize and deserialize such constructs, pointer swizzling [Eliot92] is required. This process collects all objects that are directly or indirectly referenced by the object we wish to serialize, and assigns a unique identifier to each one. In the serialized form, the pointers are replaced by the identifiers, and converted back to pointers at deserialization time.

C++ pointers provide no information on the number of items pointed to by a pointer (it can be a single item or an array of items of the same types). In order to avoid any ambiguity, DPS provides no type handlers for any pointed types. Instead, a template class *SingleRef* encapsulating a pointer is provided, in order to highlight the fact that it points to a single item. *SingleRef* also uses reference counting for managing the pointed object's lifetime. Variants of the standard STL *vector* and *map* containers allowing for pointer contents are also provided in order to allow serialization of variable-sized collections of pointed objects. DPS provides no built-in support for fixed-size arrays of pointers.

The mechanism for pointer swizzling is simple: when a pointer is encountered, the pointed object is added to a list of objects that are part of the current serialization. This list may not contain duplicate items. The pointer is then substituted with the index of the object within this list. Index 0 is reserved for NULL pointers. The serialization proceeds until all objects in the list have been serialized. This approach works for all pointer-based data structures, since objects are serialized only once even if they are referenced multiple times (e.g. in circular lists). The types of the objects in the list must also be recorded as part of the object serialization in order to enable their reconstruction.

When deserializing the data, the list of object types is recovered first, and all objects are instantiated. All objects are then deserialized in the same order as they were serialized. When pointers are encountered, the index is read from the source data stream, and the appropriate pointer is taken from the object list.

In order to support polymorphic types properly, pointed objects must include a series of virtual functions for read, write and identification operations. These functions simply call their non-virtual counterparts. They are generated by the *CLASSDEF* macro. When objects are used only by value and never pointed to, the virtual functions can be bypassed by using the *SIMPLECLASSDEF* macro instead of *CLASSDEF*.

C.6.4. Object instantiation

When deserializing a complex object that references multiple other objects, initially only the types of the objects are known. To instantiate the corresponding objects, the class factory described in section C.3 is used.

Using strings for identifying classes is very inefficient, since string operations are slow. Therefore the class factory also generates an ordered table of all the types in the application, allowing the use of indexes into this table to identify types instead of their names. When two applications wish to communicate, a mapping table from the source application's type table to the target application's type table is generated. A simple lookup in the table is then sufficient to map types from one application to another. This lookup table is generated only once when the communication channel is established, since types are static within C++ applications.

C.6.5. Rich data output

The serializer also supports output to rich formats, such as XML files. In such cases, it is necessary to output the names of the class members in addition to their values. These names can be provided by the reflection mechanism.

Furthermore, XML files also reflect the hierarchical structure of the objects. This requires two additional methods in the Stream classes, which indicate when the serializer enters or leaves an element:

```
virtual void enter(const char *name);
virtual void leave(const char *name);
```

In cases where high performance is critical and rich data output is not asked for, for example parallel application written with DPS, the calls to these methods can be suppressed from the source code with a C preprocessor definition.

C.6.6. Sample serializer output

In order to illustrate the output of the serializer, we present here a simple circular list.

```
class Circular : public dps::AutoSerial
{
  CLASSDEF(Circular)
  MEMBERS
    ITEM(std::string,value)          // A string
    ITEM(SingleRef<Circular>,next)  // Next item in list
  CLASSEND;

  Circular(const char *v = NULL) { value=v; }
};
```

A list containing three items of the above class is created and serialized to an XML file as follows (the head of the list is stored in the *head* pointer variable):

```
// Create the XML file writer
dps::XMLFileWriter xfw("circular.xml");

// Create our linked list
Circular *head = new Circular("first item");
Circular *two = new Circular("second item");
Circular *three = new Circular("third item");
head->next=two;
two->next=three;
three->next=head;

// Write the object to the output stream
xfw.write(head);
```

The resulting XML file contains the three items that were added to the list, followed by a *refs* section containing the types of the items in the file.

```
<?xml version="1.0" encoding="utf-8"?>
<xmlserializeddata>
 <item>
  <value>first item</value>
  <next>2</next>
 </item>
 <item>
  <value>second item</value>
  <next>3</next>
 </item>
 <item>
  <value>third item</value>
  <next>1</next>
 </item>
 <refs>
  <ref>Circular</ref>
  <ref>Circular</ref>
  <ref>Circular</ref>
 </refs>
</xmlserializeddata>
```

This is just one example of the type of output that is produced by the serializer. For performance reasons, the serializer generates only binary data when used within the DPS parallel framework.

C.7 Conclusion

The DPS library provides mechanisms for type handling, reflection and serialization to C++ applications. These mechanisms need to be provided by the library, since they are not natively supported by the C++ programming language. In order to make these mechanisms as unobtrusive as possible, a combination of template metaprogramming and macros has been used. The result is a mechanism that can reliably reflect, serialize and deserialize complex C++ objects in a heterogeneous environment involving any combination of systems and compilers. The implementation of the mechanism is independent of the DPS library, and can easily be used in standalone applications.

Bibliography

[Ackerman98] M. Ackerman, The Visible Human project, *Proceedings of the IEEE,* Vol. 86, No. 3, March 1998, pp. 504-511

[Adiga02] N.R. Adiga et al., An overview of the BlueGene/L Supercomputer, Conference on High Performance Networking and Computing, *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, 2002, pp. 1-22

[Agbaria99] A. Agbaria, R. Friedman, Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations, *Proceedings of the 8th International Symposium on High Performance Distributed Computing (HPDC-8'99)*, IEEE CS Press, August 1999

[Alexandrescu01] A. Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001, pp. 49-76

[Anderson02] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer, SETI@home: an Experiment in Public-resource Computing, *Communications of the ACM*, Vol. 45, No. 11, 2002, pp. 56-61

[Armstrong99] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, B. Smolinski, Toward a common component architecture for high-performance scientific computing, *Proceedings of the Eighth International Symposium on High Performance Distributed Computing*, 1999, pp. 115-124

[Attardi01] G. Attardi, A. Cisternino, Reflection support by means of template metaprogramming, *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, LNCS 2186, Springer-Verlag, Berlin, September 2001, pp. 178-187

[Bacci95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti and M. Vanneschi, P^{3}L: A structured high level programming language and its structured support, *Concurrency Practice and Experience*, Vol.7, No.3, May 1995, pp. 225-255

[Bacon94] D.F. Bacon, S.L. Graham, O.J. Sharp, Compiler transformations for high-performance computing, *ACM Computing Surveys*, Vol. 26, Issue 4, December 1994, pp. 345-420

[Baratloo95] A. Baratloo, P. Dasgupta, Z.M. Kedem, Calypso: A Novel Software System for Fault-Tolerant Parallel Procssing on Distributed Platforms, *Proceedings of the International Symposium on High-Performance Distributed Computing*, 1995, pp. 122-129

[Batchu01] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, M. Apte, MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing, *Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid*, Melbourne, Australia, 2001

[Bertrand05] F. Bertrand, R. Bramley, D.E. Bernholdt, J.A. Kohl, A. Sussman, J.W. Larson, K.B. Damevski, Data Redistribution and Remote Method Invocation in Parallel Component Architectures, *Proceedings of the International Parallel and Distributed Processing Symposium 2005*, p. 40b

[Bhargava88] B. Bhargava, S.R. Lian, Independent Checkpointing and Concurrent Rollback for Recovery – an Optimistic Approach, *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 1988, pp. 3-12

[Boehm93] H. Boehm, Space efficient conservative garbage collection, *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993, pp. 197-206

[Bosilca02] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. N´eri, A. Selikhov, MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes, *Proceedings of the High Performance Networking and Computing Conference (SC2002)*, Baltimore, USA, November 2002, pp. 1-18

[Botorog96] G. H. Botorog, H. Kuchen, Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming, *Proceedings of the 5th International Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, 1996, pp. 243-252

[Bouteiller03] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezick, P. Lemarinier, F. Magniette, MPICH-V2, a fault-tolerant MPI for volatile nodes based on pessimistic sender based message logging, *Proceedings of the High Performance Networking and Computing Conference* (SC2003), November 2003, pp. 25

[Brooks05] W. Brooks, *Large-Scale NASA Science Applications on the Columbia Supercluster, Proceedings of the International Supercomputer Conference 2005 (ISC2005)*, June 2005

[Burke86] M. Burke, R. Cytron, Interprocedural dependence analysis and parallelization, *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, 1986, pp. 162-175

[Chakravorty04] S. Chakravorty, L.V. Kale, A fault tolerant protocol for massively parallel systems, *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, April 2004, pp. 212-219

[Chamberlain00] B.L. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder, W.D. Weathersby, ZPL: A Machine Independent Programming Language for Parallel Computers, *IEEE Transactions on Software Engineering*, Vol. 26, No. 3, March 2000, pp. 197-211

[Chen97] Y. Chen, K. Li, J.S. Planck, Clip: A checkpointing tool for message-passing parallel programs, Conference on High Performance Networking and Computing, *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, November 1997, pp. 1-11

[Chiba95] S. Chiba, A Metaobject Protocol for C++, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1995, pp. 285-299

[Choi96] J. Choi, J.J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, R.C. Whaley, The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines, *Scientific Programming*, Vol. 5, 1996, pp. 173-184

[Das97] D. Das, P. Dasgupta, P.P. Das, A New Method for Transparent Fault Tolerance of Distributed Programs on a Network of Workstations Using Alternative Schedules, *Proceedings of the Conference on Algorithms and Architectures for Parallel Processing (ICAPP'97)*, 1997, pp. 479-486

[Desprez95] F. Desprez, J.J. Dongarra, B. Tourancheau, Performance complexity of LU factorization with efficient pipelining and overlap on a multiprocessor, *Parallel Processing Letters*, Vol. 5, Issue 2, 1995

[Dongarra79] J.J. Dongarra, J. Bunch, C. Moler, G. W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, PA, 1979

[Dongarra96] J. Dongarra, S. Otto, M. Snir, D. Walker, A message passing standard for MPP and Workstations, *Communications of the ACM*, Vol. 39, No. 7, 1996, pp. 84-90

[Dongarra01] J.J. Dongarra, P. Luszczeky, A. Petitet, *The LINPACK Benchmark: Past, Present, and Future*, December 2001

[Edjlali97] G. Edjlali, A. Sussman, J. Saltz, Interoperability of data parallel runtime libraries, *Proceedings of the Eleventh International Parallel Processing Symposium*, IEEE Computer Society Press, April 1997, pp. 451-459

[ElGhazawi05] T.A. El-Ghazawi, W.W. Carlson, J.M. Draper, *UPC Language Specification (V 1.2)*, June 2005

[Eliot92] J. Eliot, B. Moss, Working with Persistent Objects: To Swizzle or Not to Swizzle, *IEEE Transactions on Software Engineering*, Vol. 18, Issue 8, 1992, pp. 657-673

[Ellis90] M.A. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[Elnozahy92] E.N. Elnozahy, W. Zwaenepoel, Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit, *IEEE Transactions on Computers*, Vol. 41 No. 5, May 1992, pp. 526-531

[Elnohazy02] E.N. Elnozahy, L. Alvisi, Y.M. Wang, D.B. Johnson, A Survey of Rollback-Recovery Protocols in Message-Passing Systems, *ACM Computing Surveys*, Vol. 34, No. 3, September 2002, pp. 375-408

[Enslow77] P.H. Enslow Jr., Multiprocessor Organization - A Survey, *ACM Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 103-129

[Franke05] B. Franke, M.F.P. O'Boyle, A Complete Compiler Approach to Auto-Parallelizing C Programs for Multi-DSP Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 16, No. 3, March 2005, pp. 234-245

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, pp. 87-95

[Geist94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994

[Gennart98] B. Gennart, *CAP Reference Manual*, Internal Report, 1998

[Gennart99] B. Gennart, R.D. Hersch, Computer-Aided Synthesis of Parallel Image Processing Applications, *Proceedings of the Conference on Parallel and Distributed Methods for Image Processing III*, SPIE International Symposium on Optical Science, Engineering and Instrumentation, Denver, USA, July 1999, SPIE Vol. 3817, pp. 48-61

[GKrellM] B. Wilson, *GKrellM – GNU Krell Monitors*, http://www.gkrellm.net

[Golub96] G.H. Golub, C.F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, 1996, pp. 94-116

[Gosling96] J. Gosling, B. Joy, G. Steele, *The Java Language Specification*, Addison-Wesley, 1996

[Goujon98] D. Goujon, M. Michel, J. Peeters, J. Devaney, AutoMap and AutoLink: Tools for Communicating Complex and Dynamic Data-structures Using MPI, *Proceedings of the 4th International Symposium on High Performance Computer Architecture (HPCA-4)*, LNCS Vol. 1362, Springer Verlag, 1998, pp. 98-109

[Gray03] G.T. Gray, R.Q. Smith, Before the B5000: Burroughs computers, 1951-1963, *IEEE Annals of the History of Computing*, Vol. 25, Issue 2, April-June 2003, pp. 50-61

[Grimshaw93] A. S. Grimshaw, Easy-to-Use Object-Oriented Parallel Processing with Mentat, *IEEE Computer*, Vol. 26, No. 5, May 1993, pp. 39-51

[Gupta99] R. Gupta, S. Pande, K. Psarris, V. Sakar, Compilation Techniques for Parallel Systems, *Parallel Computing*, Vol. 25, Nos.13-14, 1999, pp. 1741-1783

[Hall96] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, M.S. Lam, Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Computer*, Vol. 29, No. 12, December 1996, pp. 84-89

[ISO05] ISO/IEC 14882, *Standard for Programming Language C++*, Working Draft April 2005, Section 14.5.4, pp. 289-291

[Johnson87] D.B. Johnson, W. Zwaenepoel, Sender based message logging, *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, July 1987, pp. 14-19

[Kale94] L. V. Kale, B. Ramkumar, A. B. Sinha, A. Gursoy, The Charm Parallel Programming Language and System: Part I - Description of Language Features, *IEEE Transactions on Parallel and Distributed Systems*, 1994

[Kuchen02] H. Kuchen, M. Cole, The Integration of Task and Data Parallel Skeletons, *Parallel Processing Letters*, Vol. 12, No. 2, 2002, pp. 141-155

[Kuchen02b] H. Kuchen, A Skeleton Library, *Proceedings of Euro-Par 2002*, LNCS 2400, Springer Verlag, 2002, pp. 620-629

[Kumar93] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing*, Benjamin Cummings Publishing Company, Chapter 11, pp. 407-489, 1993

[Lee05] J.-Y. Lee, A. Sussman, High Performance Communication between Parallel Programs, *Proceedings of the International Parallel and Distributed Processing Symposium 2005*, p. 177b

[Leiner59] A.L. Leiner, W.A. Notz, J.L. Smith, A. Weinberger, PILOT - A New Multiple Computer System, *Journal of the ACM*, Vol. 6 , Issue 3, July 1959, pp. 313-335

[LieQuan03] L. Lie-Quan, A. Lumsdaine, The generic message passing framework, *Proceedings of the International Parallel and Distributed Processing Symposium 2003*, pp. 53-62

[Louca00] S. Louca, N. Neophytou, A. Lachanas, P. Evripidou, MPI-FT: Portable Fault Tolerance Scheme for MPI, *Parallel Processing Letters*, Vol. 10, No. 4, 2000, pp. 371-382

[MellorCrummey01] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarria-Miranda, R. Fowler, G. Jin, K. Kennedy, Q. Yi, Advanced Optimization Strategies in the Rice dHPF Compiler, *Concurrency Practice and Experience*, Vol. 14, Nos. 8-9, 2002, pp. 741-767

[Messerli99] V. Messerli, O. Figueiredo, B. Gennart, R.D. Hersch, Parallelizing I/O intensive Image Access and Processing Applications, *IEEE Concurrency*, Vol. 7, No. 2, April-June 1999, pp. 28-37

[Metaxas99] P.T. Metaxas, Optimal Parallel Error-Diffusion Dithering, *Proceedings of SPIE Vol. 3648, Color Imaging: Device-Independent Color, Color Hardcopy, and Graphic Arts IV*, 1999, pp. 485-494

[Microsoft96] Microsoft Corportation, *DCOM Technical Overview*, MSDN Library, November 1996

[Nguyen03] T-A. Nguyen, P. Kuonen, ParoC++: A Requirement-driven Parallel Object-oriented Programming Language, *Proceedings of the 8th Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2003)*, International Parallel and Distributed Processing Symposium (IPDPS'03), IEEE Press, 2003, pp. 25-33

[Notredame00] C. Notredame, D. Higgins, J. Heringa, T-Coffee: A novel method for multiple sequence alignments, *Journal of Molecular Biology*, Vol. 302, 2000, pp. 205-217

[OMG02] Object Management Group, *CORBA Components*, Formal specification 02-06-65, June 2002

[OpenMP05] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, Version 2.5, May 2005

[Pande03] V.S. Pande, I. Baker, J. Chapman, S.P. Elmer, S. Khaliq, S.M. Larson, Y.M. Rhee, M.R. Shirts, C.D. Snow, E.J. Sorin, B. Zagrovic, Atomistic Protein Folding Simulations on the Submillisecond Time Scale Using Worldwide Distributed Computing, *Biopolymers*, Vol. 68, 2003, pp. 91-109

[Petitet04] A. Petitet, R.C. Whaley, J. Dongarra, A. Cleary, *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*, Innovative Computing Laboratory, January 2004, http://www.netlib.org/benchmark/hpl/

[Petzold01] C. Petzold, *Programming Windows with C#*, Microsoft Press, 2001

[Plank95] J.S. Plank, Y. Kim, J.J. Dongarra, Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations, *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS-25)*, 1995, pp. 351-360

[Platt03] D.S. Platt, *Introducing Microsoft® .NET, Third Edition*, Microsoft Press, April 2003

[Plauger00] P.J. Plauger, A.A. Stepanov, M. Lee, D.R. Musser, *The C++ Standard Template Library*, Prentice Hall, 2000

[Press90] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1990

[Prosise99] J. Prosise, *Programming Windows with MFC*, Microsoft Press, 1999

[Richardson93] J.E. Richardson, M.J. Carey, D.T. Schuh, The Design of the E Programming Language, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 15, Issue 3, July 1993, pp. 494-534

[Saito05] Y. Saito, M. Shapiro, Optimistic Replication, *ACM Computing Surveys (CSUR)*, Vol. 37, No. 1, March 2005, pp. 42-81

[Schlichting83] R.D. Schlichting, F.B. Schneider, Fail-stop processors: An approach to designing fault-tolerant computing systems, *ACM Transactions on Computer Systems*, Vol. 1, No. 3, 1983, pp. 222-238

[Skillicorn98] D. Skillicorn, D. Talia, Models and Languages for Parallel Computation, *ACM Computing Surveys*, Vol. 30, No. 2, June 1998, pp. 123-169

[Skillicorn00] D. B. Skillicorn, S. Pelagatti, Building programs in the network of tasks model, *Proceedings of the 2000 ACM symposium on Applied computing*, Vol. 1, 2000, pp. 248-254

[Stellner96] G. Stellnet, CoCheck: checkpointing and process migration for MPI, *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, April 1996, pp. 526-531

[Strom85] R. Strom, S. Yemini, Optimistic recovery in distributed systems, *ACM Transactions on Computer Systems*, Vol. 3, No. 3, 1985, pp. 204-226

[Tamir84] Y. Tamir, C.H. Sequin, Error recovery in multicomputers using global checkpoints, *Proceedings of the International Conference on Parallel Processing*, 1984, pp. 32-41

[Tanenbaum01] A. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 2001, pp. 81-100

[Teale93] S. Teale, *C++ IOStreams Handbook*, Addison-Wesley, 1993

[Toledo97] S. Toledo, Locality of Reference in LU Decomposition with Partial Pivoting, *SIAM Journal on Matrix Analysis and Applications*, Vol. 18, No. 4, October 1997, pp. 1065-1081

[Wang93] Y.M. Wang, W.K. Fuchs, Lazy Checkpoint Coordination for Bounding Rollback Propagation, *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993, pp. 78-85

[Whaley01] R.C. Whaley, A. Petitet, J.J. Dongarra, Automated Empirical Optimization of Software and the ATLAS Project, *Parallel Computing*, Vol. 27, No. 1-2, 2001, pp. 3-35

[Wilson92] P.R. Wilson, Uniprocessor Garbage Collection Techniques, *Proceedings of the International Workshop on Memory Management*, Lecture Notes In Computer Science, Vol. 637, Springer-Verlag, September 1992, pp. 1-42

[Wolf91] M.E. Wolf, M.S. Lam, A data locality optimizing algorithm, *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*, June 1991, pp. 30-44

[Wolfe95] M. Wolfe, *High-Performance Compilers for Parallel Computing*, Addison Wesley, 1995

Biography

Sebastian Gerlach was born on February 16, 1978 in Hamburg, Germany. He spent most of his school years in Switzerland, with a short two year break in England. He graduated from high school in Nyon, Switzerland in 1995. He then studied microengineering at EPFL, obtaining his master's degree in 2000. The main topic of his masters' thesis was checkpointing in parallel striped file systems in order to create fault-tolerant data intensive applications. Following his graduation at EPFL, he continued his research at the Peripheral Systems Lab of the School of Computer and Communication Sciences of EPFL. His research topics include both high performance visualization solutions for the Visible Human dataset and fault-tolerant dynamic parallel applications. The first topic has given rise to some of the services available on EPFL's Visible Human Server web page (http://visiblehuman.epfl.ch), and the second topic to the DPS framework (http://dps.epfl.ch). He is also lecturer for the course "Peripherals" in the computer science master program.

His other computer-related interests relate to real-time 3D computer-generated imagery. As a member of the Calodox collective (http://www.calodox.org), he focuses mostly on extreme coding techniques (4k executables).

Personal Bibliography

[Evesque02] F. Evesque, S. Gerlach, R.D. Hersch, Building 3D anatomical scenes on the Web, *Journal of Visualization and Computer Animation*, J. Wiley, Vol. 13, 2002, pp. 43-52

[Gerlach00] S. Gerlach, R.D. Hersch, The Real-Time Interactive Visible Human Navigator, *Proceedings of the Third Visible Human Conference*, October 2000, pp. 25-26

[Gerlach02] S. Gerlach, R.D. Hersch, A Real-Time Navigator for the Visible Human, *IEEE Internet Computing*, Vol. 6, No 2, March-April 2002, pp. 27-33

[Gerlach02b] S. Gerlach, R.D. Hersch, Two Approaches for Applet-based Visible Human Slice Extraction, *Proceedings of SPIE Internet Imaging III*, SPIE Vol. 4672, 2002, pp. 10-19

[Gerlach02c] S. Gerlach, R.D. Hersch, Exploring Anatomic Structures with EPFL's Visible Human Server, *Proceedings of the Fourth Visible Human Conference*, October 2002

[Gerlach03] S. Gerlach, R.D. Hersch, DPS - Dynamic Parallel Schedules, *Proceedings of the 8th Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2003)*, International Parallel and Distributed Processing Symposium (IPDPS'03), IEEE Press, 2003, pp. 15-24

[Gerlach05] S. Gerlach, R.D. Hersch, Fault-tolerant Parallel Applications with Dynamic Parallel Schedules, *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'05)*, IEEE Press, 2005, p. 278b

[Gerlach06] S. Gerlach, B. Schaeli, R.D. Hersch, Fault-tolerant Parallel Applications with Dynamic Parallel Schedules: A programmer's perspective, *to be published in Dependable Systems: Software, Computing, Networks*, Lecture Notes in Computer Science, Springer Verlag, 2006

[Saroul03] L. Saroul, S. Gerlach, R.D. Hersch, Exploring Curved Anatomic Structures with Surface Sections, *Proceedings of the IEEE Visualization Conference*, IEEE Press, October 2003, pp. 27-34

[Schaeli06] B. Schaeli, S. Gerlach, R.D. Hersch, A simulator for parallel applications with dynamically varying compute node allocation, *to be published in Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'06)*, IEEE Press, 2006