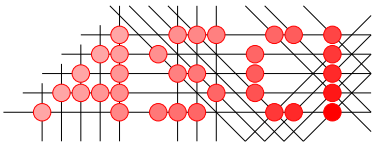
Handling complexity and change in grid computing



This work was carried out in graduate school ASCI.
ASCI dissertation series number 143.

This work was carried out in the context of Virtual Laboratory for e-Science project (ww.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ).

VRIJE UNIVERSITEIT

Handling complexity and change in grid computing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 10 mei 2007 om 10.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Małgorzata Wrzesińska

geboren te Warschau, Polen

promotor: prof.dr.ir. H.E. Bal
copromotor: dr. J. Maassen

Czyżby mi się udało?

Contents

List of Figures

List of Tables

Acknowledgments

Even though only a single name is listed on the cover of this thesis, many people have contributed to it. I would like to use this section to acknowledge these contributions.

Henri Bal and Jason Maassen were the supervisors of my PhD project. Most of the ideas presented in this thesis were inspired by the discussions with them. Apart from these countless discussions they also invested much time into reading and correcting my papers and this thesis.

Rob van Nieuwpoort is the author and implementor of the prototype Satin system. The work described in this thesis is based on his research and the current implementation of Satin is based on his prototype.

Rob van Nieuwpoort and Jason Maassen are the designers and implementors of the Ibis communication library on top of which Satin is built.

Ceriel Jacobs constantly works on keeping the Ibis and Satin source code complete, orderly and, most importantly, efficient. He corrected the countless bugs I introduced into Satin and implemented many features, for which I could not find time or simply was not skilled enough. He also helped to keep to my source code consistent in the most busy period of my PhD.

Niels Drost drank hectoliters of coffee with me while discussing many ideas described in this thesis. He is also the author of Zorilla, which was used to implement the adaptivity component from chapter 4.

Kees Verstoep is the administrator of the DAS-2 supercomputer which was used for the experiments presented in this thesis. Kees was always very helpful when I had problems with node reservations. He also wrote the SAT solver application I use in chapter 5.

Maik Nijhuis wrote the first version of the Barnes-Hut application which was later optimized by Ceriel and Rob and used in chapter 4 of this thesis.

Kris Borg implemented the first version of checkpointing described in chapter 3. This work was his Master's project.

Thilo Kielmann co-supervised the checkpointing project together with myself. Thilo also inspired the adaptivity work described in chapter 4.

Mathijs den Burger allowed me to use his traffic shaper for the experiments in chapter 4.

Ana Oprescu helped with the performance evaluation section in chapter 3.

The cover of this thesis was designed by Wouter Gransbergen (Miś) and the background photo was taken by Grumpy.

Chapter 1

Introduction

1.1 Motivation and goals

Grid environments integrate heterogeneous and geographically-distributed computing resources into a single system. Many applications can benefit from such environments, for example collaborative applications, which enable remote collaborations and sharing of computational resources or data-intensive applications, which process data located on geographically distributed resources. In this thesis, we focus on another interesting class of applications: *distributed supercomputing* applications. Distributed supercomputing applications use computational grids to solve computational challenges that could not be tackled on a traditional parallel systems. Grids provide computational power many times larger than that of a traditional supercomputer. However, the complexity of Grid environments also is many times larger than that of traditional parallel machines. Grid environments are inherently heterogeneous. Grids consist of machines with various processor architectures and various operating systems. Processor speeds vary dramatically. Finally, the quality of network connections varies from low-latency Local Area Networks (LANs) to high-latency and possibly low-bandwidth Wide Area Networks (WANs). Grid environments are also inherently dynamic. The availability of resources is constantly changing. Processors may crash or become unavailable because they are claimed by a higher-priority application or because a reservation has ended. New processors may become available. Also, the load on the resources, both network links and processors, is constantly changing.

In order to achieve good performance, grid applications need to be able to tolerate high wide-area latencies (i.e., they need to be latency insensitive) and possibly low bandwidths. They need to be portable (i.e., able to run on multiple architectures without the need of recompilation) and able to efficiently utilize processors with various speeds (i.e. the fast processors should not have to wait for the slow ones. Finally, they need to adapt to dynamic characteristics of the environment.

Writing grid-enabled applications is therefore an inherently complex task. The programmer does not only need to have deep understanding of the application problem domain, but also of the complex parallel and distributed programming issues such

as: optimizing the inter-processor communication, load balancing, fault tolerance, adaptivity etc. Because of this complexity, few grid-enabled applications have been developed until now and the tremendous power of grid environments is still mostly unused. Therefore, the process of creating grid applications needs to be simplified.

We believe that this goal can be achieved with grid programming *frameworks* (high-level grid programming environments). A framework is a set of tools (such as compiler, runtime system, libraries etc.) that forms a *layer of abstraction* between the application and the low-level grid infrastructure. Frameworks present a programmer with a high-level programming model that abstracts away the details of the underlying platform. Because the programming model is high-level, it does not support all possible applications, but only a certain *class* of applications. However, the advantage of narrowing the supported application set is that most of the grid related issues can be resolved automatically by the framework software. In contrast, low-level programming environments (e.g., message-passing environments such as MPICH [112]) support a wider range of applications, but the application programmer is responsible for dealing with grid issues.

In this work we focus on the class of divide-and-conquer applications. Divide-and-conquer is a popular and efficient paradigm for writing grid applications [32, 139]. Divide-and-conquer algorithms operate by splitting the problem into subproblems and then solving them recursively. The divide-and-conquer paradigm is a generalization of the popular master-worker paradigm. The task graph of a divide-and-conquer application is *hierarchically* structured. Therefore, such applications can be executed with excellent communication locality in grid environments, which are usually also hierarchical: they consist of multiple clusters or supercomputers with low-latency intra-cluster communication and high-latency inter-cluster links.

The divide-and-conquer paradigm has broad applicability in many fields such as astrophysics, bioinformatics, computational geometry, numerical methods, games and other search and optimization problems. Also, all master-worker computations can be expressed in the divide-and-conquer model.

In earlier work by Rob van Nieuwpoort [175] a prototype divide-and-conquer framework called *Satin* was designed and implemented. The Satin framework consists of a compiler and a runtime system, both written entirely in Java. Java is also used to write applications with Satin. This allows the application to run over heterogeneous architectures without the need of recompilation (thanks to Java's 'write once, run anywhere' property). Satin extends the sequential Java language with two simple divide-and-conquer primitives: *spawn* and *sync*. The programmer writes the application in a recursive way and annotates the *sequential* code with those primitives to create a grid application. The Satin compiler generates the necessary communication and load-balancing code. Satin uses a grid-aware load-balancing strategy called Cluster-aware Random Work Stealing (CRS) [176] which allows Satin applications to run very efficiently in a wide-area setting [178]. Also, because work stealing is a *dynamic* load balancing strategy [175], it allows efficient usage of processors with various speeds and/or variable load.

The combination of Java and CRS allows Satin to resolve a number of grid issues, namely the heterogeneity of processor architectures, the heterogeneity of processor

speeds and large wide-area latencies. However, there is still a large number of problems that need to be solved before Satin becomes a mature grid programming framework. In the following sections, we will describe those problems and sketch the solutions which will be presented in more detail in the remaining part of this thesis. The result of the work presented in this thesis is a full-fledged, mature grid programming framework.

1.2 Heterogeneity and change

An important problem in grid computing is *resource selection*: which resources and how many resources should we use to achieve good performance? Even in traditional parallel environments (single cluster or supercomputers) finding the optimal number of processors is a difficult task and is often solved in a trial-and-error fashion. In grid environments, this problem is an order of magnitude harder because of the heterogeneity of resources. Even though Satin can handle the heterogeneity of processor architectures and speeds and can run efficiently on high-latency networks, there are still combinations of resources that will result in very poor performance. For example, when some very slow processors are used, the performance gain they might provide will not outweigh the load-balancing and communication overhead they introduce. Also, if bandwidth on a certain link is lower than a certain minimal bandwidth (which is different for each application) the performance of the application dramatically deteriorates. Finally, using more processors than the application's level of parallelism allows will result in poor resource utilization.

Another problem is the dynamic characteristics of the grid environment. The availability of resources constantly changes. Grids are inherently more unreliable than traditional parallel computers or clusters. The number of processors and network links is much larger and therefore the mean-time-to-failure becomes much shorter. There is no centralized control, so (a part of) our resources can be turned off for maintenance or simply given to another user. The resources are shared by many users, so they can become overloaded. To survive in such an environment, the application needs to be *fault tolerant*, that is, able to continue working in the presence of processor and network failures. In order to not only survive but also achieve good performance, the application needs to *adapt* to changing conditions. This involves *malleability*, which is the ability to change the number of processors used on the fly and *migratability*, which is the ability to transfer to another set of resources during the application run.

In chapters 3 and 4 we will discuss the solutions to those problems. First, we discuss the question of providing fault tolerance, malleability and migratability to divide-and-conquer applications. In chapter 3, we will present a simple algorithm that provides fault tolerance, malleability and migratability to divide-and-conquer applications. Using this algorithm, the applications can handle joining/leaving processors and migrate with an overhead that is close to zero.

In chapter 4, we will show how to use malleability to provide a solution to the adaptation and the resource selection problem. Existing solutions to those problems require providing a *performance model* for an application. Such a performance model

is used to *predict* the running time of the application on a given set of resources. Various resource sets are compared using the performance model and the resource set which yields the shortest runtime is selected for execution. To provide adaptivity, this resource selection phase is repeated during application execution, either at regular intervals or when performance degradation is detected. Constructing performance models, however, is inherently difficult. Creating such a model requires expertise which an application programmer might not have. In chapter 4, we discuss an alternative approach to application adaptation and resource selection. We start an application on *any* set of resources. During the application run, we collect statistics about the run and use them to deduce the resource requirements of the application. Next, we *adjust* the resource set the application is running on by adding or removing nodes. Thus, we are using malleability to achieve adaptivity. This approach does not necessarily result in the optimal resource set. However, it allows avoiding various performance bottlenecks, such as slow WAN links or overloaded processors. We demonstrate the working of this approach in various scenarios typical for grid environments and show that significant performance improvements can be achieved.

1.3 Data sharing in dynamic environments

Divide-and-conquer is a paradigm with a broad range of applications. However, an important disadvantage is the lack of global state. The only way of sharing data between tasks is by explicit parameter passing and returning results. This model turns out to be insufficient for many applications. One class of such applications consists of programs that pass large data structures as parameters. With pure divide-and-conquer, those large parameters need to be copied each time a task is executed remotely (stolen), while copying the parameters once and reusing them later would be more efficient. Another class of applications consists of programs that need to share data between independent tasks. In pure divide-and-conquer, this form of data sharing is not possible. Branch-and-bound applications belong to this class. Sharing the best known solution between all the processors taking part in the computation allows pruning large parts of the search tree. Another example is game-tree search where a transposition table is shared to avoid evaluating the same position twice.

In chapter 5, we investigate the possibility of extending the divide-and-conquer model with a shared data abstraction. We propose a divide-and-share model: the divide-and-conquer model extended with a shared data abstraction – shared objects. Implementing a shared data abstraction on the Grid is a challenging problem. Providing strong consistency while maintaining high performance is infeasible even on tightly connected systems like clusters of workstations. In grid environments, it is even harder due to large wide-area latencies and due to the fact that grid environments are inherently dynamic. Luckily, many applications can tolerate weaker consistency models. In fact, only applications that can tolerate weaker consistency will be able to efficiently run in grid environments. Many consistency models have been proposed but none of them are suitable for divide-and-conquer grid applications. As we will explain in more detail in chapter 5, they are either too expensive to imple-

ment in grid environments, or do not fit the needs of our applications. Therefore, we will introduce a new, relaxed consistency model, which we call *guard consistency*. With guard consistency, the programmer can define the consistency requirements of an application by means of boolean *guard functions*. A guard function is associated with a divide-and-conquer task and defines whether the shared data accessed by this task are in a correct state from the application's point of view. The runtime system uses an inexpensive *optimistic* protocol which allows the object replicas to become different as long as guards are satisfied. Only when a guard becomes unsatisfied, does the runtime system bring the local replica into consistent state which is a potentially expensive operation.

Using the divide-and-share model we implement a number of new applications and evaluate them in a real grid environment. We demonstrate that our applications can achieve high efficiencies in such environments.

1.4 Contributions

The starting point for this work was a prototype divide-and-conquer framework implemented by Rob van Nieuwpoort. In this thesis, we will show how we turned it into a mature, full-fledged grid computing environment. The contributions made in this thesis can be summarized as follows:

1. We have designed and implemented a set of algorithms that provide fault tolerance, malleability and migratability to divide-and-conquer applications. The resulting system can handle a vast variety of scenarios typical for the Grid:

   - crashing processors, including a total crash can be handled
   - processors joining and leaving an on-going computation can be handled with high efficiency
   - an application can be efficiently migrated
   - an application can be stopped and restarted later on a possibly different set of resources

2. We propose a novel approach to resource selection and adaptation that does not require constructing analytical performance models for applications. Our approach improves application performance in many different situations that are typical for grid computing. It handles all of the following cases:

   - automatically adapting the number of processors to the degree of parallelism in the application, even when this degree changes during the computation
   - migrating (part of) a computation away from overloaded resources
   - removing resources with poor communication links that slow down the computation

- adding new resources to replace resources that have crashed

3. We have improved the applicability of the Satin framework by extending the divide-and-conquer programming model with a shared data abstraction: shared objects. Shared objects provide a novel consistency model called guard consistency. We have shown that a shared data abstraction can be implemented efficiently in dynamic grid environments.

1.5 Outline of this thesis

The rest of this thesis is structured as follows. In chapter 2, we classify and review existing grid programming environments. Further, we describe the prototype Satin framework designed and implemented by Rob van Nieuwpoort. We outline the issues that need to be resolved to turn the prototype Satin into a full-fledged, mature grid programming environment. Finally, we compare both the prototype and the full-fledged Satin with other grid programming environments. In chapter 3, we will present an algorithm that provides fault tolerance, malleability and migratability to divide-and-conquer applications. We will describe its implementation in Satin and its performance evaluation. In chapter 4, we will address the problems of resource selection and adaptation to changes in grid environments. We will present a simple approach to those problems and we will evaluate it in a number of scenarios typical for grid environments. In chapter 5, we will show how we can improve the applicability of the Satin framework by extending its programming model with a shared-data abstraction. We will draw our conclusions in chapter 6.

Chapter 2

Context: grid programming environments

2.1 Introduction

In this chapter, we review the related work. We propose a classification of the existing grid programming environments (GPEs) and discuss the most important of those tools. Further, we will describe the Satin programming environment and programming model and illustrate it with a number of code samples. We explain the Cluster-aware Random Work Stealing algorithm and briefly describe the implementation of Satin. Finally, we will compare Satin to other grid programming models. The remaining chapters will give more specific related work concerning the topics described in those chapters (fault tolerance, adaptivity and data sharing).

2.2 Grid programming environments

Programming grid applications consists of two major tasks: *application development* and *application deployment*. *Application development* consists of dividing the problem into tasks that can be done in parallel, mapping those tasks to physical processors, providing inter-processor communication and synchronization. *Application deployment* involves resource selection, discovery and reservation, spawning processes and providing file I/O. Grid programming tools can be roughly divided into two classes: tools that support application development (grid programming models) and environments that support application deployment. Typically, a grid programming model is combined with an application deployment tool to achieve full functionality. Some grid programming models (e.g., Proactive [30]) provide also application deployment functionality.

In the rest of this section, we will review a number of grid programming tools. We do not attempt to present *all* existing grid programming tools. We selected those that in our opinion have the biggest impact on the grid computing community. We will

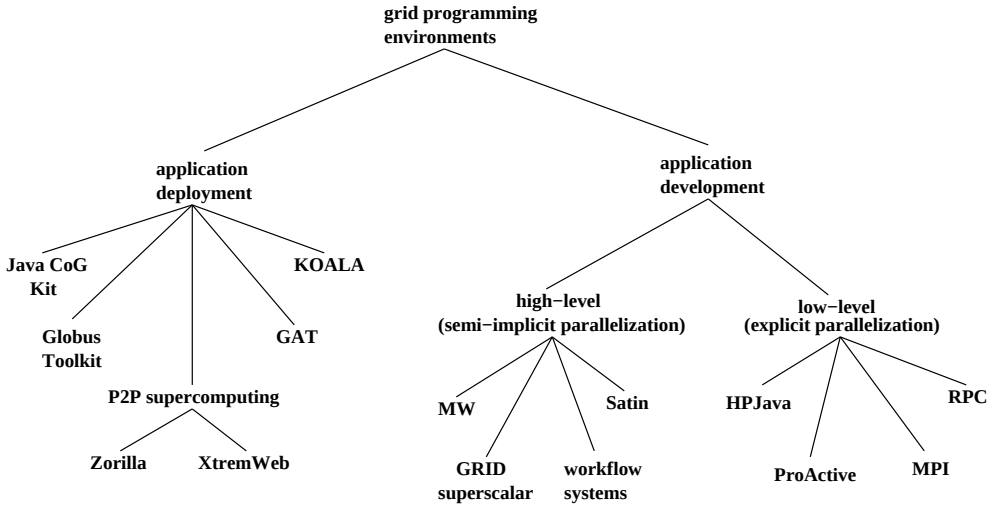


Figure 2.1: The classification of the grid programming environments

start with application deployment tools and describe the Globus Toolkit [86] which is de facto a standard in grid computing, Java Commodity Grid (CoG) Kit [180] which provides among others a Java binding to Globus tools, KOALA [136] which provides co-allocation of multiple sites, Grid Application Toolkit (GAT) [23] which can be layered on top of Globus, CoG Kit or other middleware and provides higher-level application deployment functionality, and grid middlewares based on peer-to-peer technology: Zorilla [72] and XtremWeb [54].

Next, we will describe application development tools – grid programming models. We will divide the grid programming models into high-level programming models and low-level, explicit communication models. With high-level models, the programmer only needs to be concerned with decomposing the problem into tasks that can be done in parallel. The programming environment (the compiler and/or the runtime system) will take care of low-level issues such as mapping tasks to physical processors (load balancing), inter-processor communication, fault tolerance etc. The high-level models we discuss include: grid superscalar [29], a master-worker framework (MW) and workflow systems. Explicit communication programming models typically provide only a communication abstraction. The programmer needs to not only take care of the problem decomposition but also of the low-level issues. The explicit communication models we discuss include: HPJava, MPI, ProActive and Remote Procedure Calls. The classification of all grid programming environments discussed in this chapter is shown in Figure 2.1.

2.2.1 Application deployment tools

The functionalities that application deployment tools need to provide include:

- *Resource discovery:* finding compute nodes suitable for the execution of our application.
- *Resource reservation:* reserving compute nodes, network links and possibly other resources.
- *Remote execution:* creating processes on remote resources.
- *File I/O:* Staging of the executable, input and output files. Remote file access.

Application deployment tools can be divided into low-level middleware that *exposes* the complexity of the grid to the programmer and higher-level tools that *hide* the grid complexity. The Globus Toolkit and Java CoG Kit belong to the former class while the Grid Application Toolkit belongs to the latter group.

Globus

Globus Toolkit is a set of libraries and programs that address common problems that occur when building grid applications [86]. Globus is becoming a standard in grid computing. The most important components on the Globus Toolkit are:

- The Monitoring and Discovery Service (MDS) which provides information about grid resources. MDS can be used by applications for resource discovery.
- The Globus Resource Allocation Manager (GRAM) which provides resource allocation and remote execution functionalities.
- The Globus Access to Secondary Storage (GASS) which provides access to remote files. GASS is typically used for executable, input and output file staging.
- GridFTP which provides data transfer functionality.

The Globus Toolkit provides relatively low-level support for grid programming, i.e. it *exposes* the complexity of the grid to the programmer instead of *hiding* it. The programmer must be aware of many details of the underlying platform, for example, he must explicitly state which local resource managers have to be used (e.g., PBS or Condor) when allocating resources or he must select the appropriate file transfer protocol (e.g., FTP, HTTP etc.).

Java CoG Kit

The Java Commodity Grid (CoG) Kit provides access to grid services for Java applications. Java CoG Kit is a mapping between Java and the Globus Toolkit. Therefore, Java CoG Kit provides similar functionality as the Globus Toolkit: resource management and remote execution, file I/O and information services. Additionally, CoG provides a number of simple GUI components that can be used as building blocks for grid portals. CoG has a layered architecture (similar to the GAT below), which allows shielding the application programmer from the constant changes the Globus Toolkit is undergoing.

KOALA

An important problem of tools such as Globus Toolkit or Java CoG Kit is the lack of *co-allocation*, that is, the ability to schedule an application on multiple sites (clusters or supercomputers) simultaneously. For example, using the Globus Toolkit, the programmer can submit an application to multiple sites, but there are no guarantees that all parts of the application will be started at the same time.

This problem is addressed by the KOALA scheduler [136]. KOALA builds on top of the Globus Toolkit – it uses Globus tools to submit jobs to the individual execution sites and to stage in files. KOALA makes sure that all job components located on different sites start simultaneously. To achieve this goal, KOALA repeatedly tries to claim processors. If not enough idle processors are available on one or more sites, claiming is repeated until successful. This strategy can be optimized if a site supports advance reservations.

GAT

Grid Application Toolkit (GAT) [23] provides a simple API to grid applications. While Globus and CoG Kit expose the complexity of the grid to the application programmer, the GAT hides the details of the underlying platform. GAT can be layered *on top* of the lower-level grid middleware such as Globus, as will be explained below. The GAT consists of the following subsystems:

- Resource Management Subsystem allows the application to discover resources, reserve them and submit and manage jobs. An important component of this subsystem is the Resource Broker. The Resource Broker can find resources based on the hardware and software requirements specified by the application programmer (e.g., the amount of memory, minimal CPU speed, operating system). The Resource Broker can also reserve the resources and spawn remote processes. The application programmer does not need to be concerned about details such as local resource managers types. Such issues are resolved automatically by the GAT Resource Broker.

- File Subsystem provides the application with access to files. Using this subsystem the application can create, destroy, move, read or write files. The API is based on POSIX and is very simple to use. The application programmer needs only to specify the file name and location and the GAT will take care of selecting the appropriate access protocol (e.g., FTP, HTTP, GridFTP etc.) and automatically optimize the adjustable parameters based on the available information about the environment. The File Subsystem also provides a *logical file* abstraction. A logical file is a set of file replicas that are geographically distributed. If an application attempts to use a logical file, the GAT will automatically select the closest replica.

- Monitoring and Event Subsystem provides utilities for application and grid resource monitoring.

- Information Exchange Subsystem which allows advertising and searching for application metadata.

The architecture of GAT is based on the principle that the API layer should be *independent* of the underlying middleware. GAT features a three-layer architecture: the API layer, the GAT engine layer and the GAT adaptors layer. GAT adaptors are bindings of the GAT API to various grid middlewares, e.g. Globus, UNICORE [11], Zorilla. GAT adaptors are dynamically interchangeable at runtime. The GAT engine dispatches API calls to the adaptor layer. This layered architecture ensures that applications using GAT can run *without modifications* on top of various grid middlewares. The applications are also immune to changes in the grid middleware.

Peer-to-Peer Supercomputing

Peer-to-peer supercomputing middlewares are an alternative to traditional deployment tools. Peer-to-peer supercomputing middlewares are characterized by the lack of centralized components. Therefore, they are inherently more resilient to failures and easier to set up and maintain than traditional, centralized tools.

Zorilla [72] is one such grid middleware based on peer-to-peer technology. Zorilla implements all functionalities needed by grid applications in a fully decentralized fashion. Those functionalities include resource discovery and reservation, remote process creation and file staging. Zorilla does not provide remote file access.

The Zorilla system consists of a number of *Zorilla nodes* which form a peer-to-peer network. Nodes can be added and removed at any moment. A grid application directs its requests to its local Zorilla node which cooperates with other nodes to grant the requests. Zorilla is implemented entirely in Java and provides a Java API to grid applications.

Another example of a peer-to-peer supercomputing middleware is XtremWeb [54]. XtremWeb has a three-tier architecture: it consists of *clients*, *workers* and the *coordination service* which mediates between clients and workers. The coordination service accepts task requests from clients and launches the tasks on the available workers.

2.2.2 Application development tools

Application parallelization can be classified into three approaches: *implicit*, *explicit* and *semi-implicit* [159]. With *implicit* parallelization, the programmer writes a *sequential* application which is automatically parallelized by the environment. Automatic parallelization is not used in grid computing because it is hard to get satisfactory performance with this approach.

With *semi-implicit* parallelization, the programmer identifies the parts of the problem which can be solved in parallel. However, the environment takes care of mapping tasks to physical processors, load balancing and inter-processor communication. The semi-implicit approach is very popular in grid computing. It allows achieving high-performance while hiding most of the grid complexity from the programmer. The programmer is provided with a high-level and easy to use programming model. Examples of environments supporting the semi-implicit approach are: grid superscalar [29] (a

form of fork-join or divide-and-conquer parallelism), MW [95] (a master-worker framework), workflow systems and our Satin framework (divide-and-conquer). Below, we will refer to those environments as *high-level* programming models or *frameworks*.

With *explicit* parallelization, the programmer is responsible not only for identifying work that can be done in parallel, but also for mapping the tasks to physical processors, load balancing and communications. Examples are: HPJava [120], MPI [96], ProActive [30] and Remote Procedure Calls [154]. Environments that support this approach typically provide only some communication abstraction. Additionally, some implementations of MPI provide transparent fault tolerance and/or migration [106], however, no grid-enabled implementation currently provides this functionality. ProActive provides migration support and transparent fault tolerance. Below, we will refer to those models as explicit communication models.

For each programming model, we will discuss a number of non-functional properties that are vital in grid environments:

- *Performance:* One of the major driving forces behind grid computing is achieving higher performance than on traditional parallel systems. However, achieving high performance in grid environments is a challenging task which requires complex techniques, such as latency hiding or dynamic load balancing. Typically, high-level programming environments apply such techniques automatically while explicit communication models require the programmer to take care of performance. On the other hand, the explicit communication models, by giving the programmer full control over performance optimizations, often allow a more efficient implementation.

- *Ease of use:* A grid programming environment should hide as much grid complexity from the programmer as possible. High-level programming models are clearly easier to use than explicit communication models as they relieve the programmer from dealing with complex issues such as inter-process communication, load balancing, fault tolerance etc. Explicit communication models require the programmer to deal with such issues explicitly.

- *Applicability:* It is important that a grid programming environment supports a broad variety of applications. High-level programming models typically require the application programmer to use a specific programming paradigm which might not be suitable for all applications. Explicit communication models can be used for any type of application.

- *Support for fault tolerance, malleability, migratability:* Fault tolerance, malleability and migratability are essential features of a grid application. On systems consisting of hundreds or thousands of machines, the mean-time-to-failure may become shorter that the lifetime of an application. Moreover, grid environments lack centralized control and situations in which part of the computing resources is suddenly rebooted or claimed by a higher-priority application are not rare. Therefore, without support for fault tolerance, malleability and migration, the chance that a grid application would ever complete would be small.

High-level programming models typically provide transparent support for fault tolerance, malleability and migration. Explicit communication models often require the programmer to take care of those issues.

- *Adaptivity:* Grid environments are inherently dynamic. Not only the availability of resources changes constantly, but also the performance characteristics of available resources vary. On time-shared machines the processors may become overloaded by another, higher-priority application. Also network links may become overloaded and the available bandwidth may decrease dramatically. In order to achieve a reasonable performance, an application constantly has to adapt to changes in the grid environment. The adaptation support may be provided by the programming environment or may be added by the application programmer. Currently, few programming environments and applications have adaptation support.

- *Portability:* Grids are inherently heterogeneous. Therefore, a grid programming environment should not be tied to any specific platform. It should abstract away various platform-specific issues from the application. Another important issue is the programming language supported by a grid programming environment. Therefore, languages such a Java are becoming popular in grid computing. Thanks to the virtual machine technology, Java applications can run on heterogeneous architectures without the need of recompilation and porting. Thanks to JIT technology, the performance of Java applications is currently comparable with the performance of C applications [51].

Grid superscalar

When programming with the grid superscalar model [29], the programmer has to structure the application as a set of possibly repetitive, sequential tasks. Such tasks can be executed in parallel on the grid. The programmer must provide an *IDL file* specifying which tasks should be considered for a parallel execution. The IDL used in grid superscalar is based on CORBA IDL.

Each task operates on a set of files. Tasks that operate on the same file can have a data dependency. The grid superscalar compiler analyzes the data dependencies automatically. The grid superscalar runtime system maintains a graph of tasks. Edges of this graph denote data dependencies. When a task is completed, it is removed from the graph and the graph is searched for tasks with no incoming edges (i.e., no data dependencies). Such tasks are submitted for execution. The user is required to specify a file with a list of nodes that will be used for the execution. The runtime system uses the Globus Toolkit (see section 2.2.1) to execute tasks on those servers. However, the core of grid superscalar is independent of the grid middleware and can be combined with any software from section 2.2.1.

- *Performance:* No extensive performance evaluation of the grid superscalar system has been performed yet. In [29] experiments on up to 8 CPUs (on 2 nodes) are reported. A 6-fold speedup was the maximal speedup achieved on this

testbed. At the moment, it is not clear how much performance can be expected from the grid superscalar applications. However, since a GRAM call is performed to spawn each task, fine-grained applications will not perform well, since the cost of the GRAM call will not be amortized by the execution time of the task. Therefore, grid superscalar is only suitable for coarse-grained applications.

- *Ease of use:* Grid superscalar provides a high-level programming model which hides most of the grid complexity and parallel-programming issues from the programmer.
- *Applicability:* The fork-join/divide-and-conquer parallelism supported by the grid superscalar is applicable to a large class of problems. However, only *coarse-grained* parallel applications can be implemented efficiently with grid superscalar, as explained above.
- *Fault tolerance, malleability, migration:* Currently grid superscalar does not support fault tolerance, malleability and migration. Adding transparent support for fault tolerance is planned in the future.
- *Adaptivity:* Currently, grid superscalar does not provide support for adaptation. In the future, a scheduling policy that takes into account dynamic information on the system load will be used.
- *Portability:* Grid superscalar applications are written in C++ or Perl. Applications written in C++ need to be recompiled for each architecture/operating system and therefore their portability is limited. Perl is an interpreted language and therefore applications written in Perl can be run on different systems without the need of recompilation, as long as a Perl interpreter is available on a given system.

MW – a master-worker framework

MW [95] is a framework for writing grid-enabled master-worker applications. In master-worker applications, a single process called the master divides the problem to be solved into independent tasks and dispatches those tasks to the worker processes. After solving a task, a worker process returns the result to the master and requests a new task. The master-worker paradigm is very popular in grid computing. Since the tasks are independent, little communication is needed and high performance can be achieved even on wide-area networks.

The MW API is extremely simple: the programmer needs to provide only a small number of functions: a function to split up work, worker initialization routine, a function performing the actual task etc. The runtime system takes care of load balancing, inter-processor communication and fault-tolerance. MW also abstracts an Infrastructure Programming Interface (IPI) which allows to port the framework to different Grid middleware. MW was implemented on top of Condor [169] and PVM [162]. In the future, it will be ported to Globus Toolkit [86]

- *Performance:* Master-worker applications typically achieve high performance on the grid. MW has been reported to achieve high efficiencies. It has been used to solve a combinatorial optimization problem on a heterogeneous, wide-area testbed consisting of 502 processors in 7 clusters. A parallel efficiency of 80% was achieved on this testbed.

- *Ease of use:* MW provides a very high-level programming model and is therefore extremely easy to use. The application programmer is shielded both from the complexity of the grid environment and from complex parallel programming issues such as load balancing and communication.

- *Applicability:* MW supports only embarrassingly parallel applications. However, many useful problems exhibit this structure.

- *Fault tolerance, malleability, migration:* MW transparently handles worker crashes. If a worker fails, the task executed by this worker is re-assigned to another worker by the runtime system. A failure of the master has to be treated in a special way. MW offers a feature to checkpoint the state of the master. The programmer, however, needs to provide functions that write and read the state of the master. MW is also malleable. Leaving workers are handled using the fault-tolerance mechanism. Joining workers receive tasks from the work queue of the master.

- *Adaptivity:* Master-worker applications use dynamic load-balancing which allows them to adapt to varying processor speeds: slower processors get fewer tasks to process.

- *Portability:* MW applications are written in C++ and they have to be compiled separately for each platform, which limits their portability.

Workflow systems

Grid workflows are meta-applications running on the computational grid. A workflow is an aggregation of multiple sequential or parallel applications (called components in this context) which cooperate by passing files or data. The simplest workflow is a pipeline in which components are arranged in a chain and each component receives data from the previous component in the chain, processes the data and passes it to the following component. In general, a workflow is a directed graph of components, in which edges express data dependencies between the components.

Workflow systems are environments which allow building workflows out of individual components. Workflow systems often provide a graphical user interface that allows rapid development of workflow applications. Alternatively, the programmer can use technologies such as XML to define the dependencies between the components. Workflow systems automatically map workflow components onto the available grid resources. This mapping is performed in such a way that the runtime of the workflow application is minimized and/or other user constraints are met (e.g., the accuracy of the result). Workflow systems typically use application development tools,

such as the Globus Toolkit or GAT, to find the appropriate grid resources, schedule and execute workflow applications.

A vast number of workflow systems exist, for example: DAGMan [166], Pegasus [66], Triana [168], ICENI [134], GridAnt [26], GridFlow [53], Gridbus workflow [187], Kepler [25], Taverna [141], Askalon [79], VLAM-G [13], GrADS [172] and ASSIST [20] (see [188] for a detailed overview of many of those systems).

- *Performance:* Workflow systems automatically map workflow components onto the Grid to maximize the performance of the workflow. To achieve this goal, static and/or dynamic information about the grid environment (e.g. the number of available processors, estimated data transfer times etc.) is used.

- *Ease of use:* Workflow systems are extremely easy to use. The application programmer needs to specify only the data dependencies between workflow components. The programmer does not need to explicitly deal with the complexity of the grid environment.

- *Applicability:* The workflow model is suitable only for coarse-grained parallel applications.

- *Fault tolerance, malleability and migration:* Most workflow systems support fault tolerance. A vast variety of techniques is used. Most commonly, fault tolerance is provided transparently to the application programmer. For example, a failed component can restarted on the same or alternative resource. Components can be also replicated on multiple resources or checkpointed. Some systems provide support for migration, for example the GrADS systems.

- *Adaptivity:* Most workflow systems maps workflow application to grid resources statically, i.e., after the execution of the application has started, the mapping cannot be changed. Such systems, therefore, do not support adaptivity. Pegasus [66] handles dynamic changes in grid environment using *just-in-time* scheduling. With just-in-time scheduling, rather than mapping all components at once, each component is mapped to a physical resource only after all its data dependencies have been resolved, that is, after all components it depends on have finished execution. Just-in-time scheduling performs better in dynamic environments than static scheduling. However, once a component is started it cannot be remapped to a different resource, which can result in poor performance. GrADS [172] and ASSIST [20] support adaptivity by monitoring performance of the application components and migrating them to better resources if a performance degradation is required. Those systems assume that a *performance model* (i.e., a mathematical formula that allows to predicts the runtime of a component of a given resource) is known for each component.

- *Portability:* The portability of workflow systems varies greatly. Many of those systems are based on the Java technology which enhances their portability.

HPJava

HPJava [120] is a Java-based framework supporting *data-parallel* programming style. It extends sequential Java with support for *distributed arrays*: arrays that are physically distributed over the memories of the participating processors. The programmer manipulates those arrays using high-level constructs such as the *overall* construct which denotes a distributed, parallel loop.

The programming model of HPJava has been inspired by the High Performance Fortran (HPF) programming model [85] and many constructs look similar to the constructs used in HPF, for example *overall* resembles HPF's *forall*. In fact, the programming model provided by HPJava is lower-level than that of HPF. The main difference between HPJava and HPF is that with HPJava a process can only access locally held elements of distributed arrays. If a process needs to access an element held by another processor, explicit communication must take place. With HPF, processes are allowed to access any element of a distributed array and the compiler takes care of the communication.

HPJava provides a communication library called *Adlib* which implements collective communication primitives. Those primitives are expressed in terms of distributed-array operations. Some examples of operations provided by Adlib are: *remap* which changes the mapping of a distributed array to processors, *shift* which copies a given array to a new array and shifts all elements by a given number of positions, and *maxval* which returns the maximum element of a given distributed array.

Currently, distributed implementations of the HPJava collective communication rely on availability of native communication interfaces

- *Performance:* No extensive performance evaluation of HPJava has been performed. In [120] experiments on up to 36 CPUs (in as single, homogeneous cluster) and speedups up to 17 are reported. However, since HPJava is an explicit communication programming model, the application programmer will have to take the responsibility for grid-specific optimizations, such as dynamic load balancing and latency hiding.

- *Ease of use:* HPJava offers a relatively low-level programming model and therefore burdens the programmer with tasks such as load balancing and inter-process communication. Programming the communication is somewhat simplified by the array primitives provide by the Adlib communication library. Also, the programmer has to explicitly deal with some grid-related issues.

- *Applicability:* HPJava supports data-parallel applications. Many important scientific problems can be programmed in this style.

- *Fault tolerance, malleability, migration:* HPJava currently does not support fault tolerance, malleability or migration.

- *Adaptivity:* HPJava does not provide support for adaptivity. Adaptive features need to be programmed by the application programmer.

- *Portability:* The use of Java technology enhances the portability of HPJava application. Thanks to Java's 'write once, run anywhere', HPJava's applications can be run unmodified in heterogeneous environments. However, currently the distributed-memory implementation of HPJava relies on native communication interfaces (MPI or LAPI) which severely reduces the portability of the system. A pure Java implementation is planned in the future

MPI

Explicit *message passing* is a popular parallel programming paradigm. Message-passing applications are structured as a set of processes communicating via messages. The Message Passing Interface (MPI) [71] is a standard that defines the syntax and semantics of a set of communication primitives useful for that type of applications. MPI features synchronous and asynchronous point-to-point communication and various forms of collective communication, e.g. broadcast, scatter, gather and all-to-all exchanges. MPI is typically used for SPMD (Single Program Multiple Data) style programs. In SPMD programs, all processors execute the same program on a different part of the data.

Multiple implementations of the MPI standard exist. MPICH-G2 [112] is a grid-enabled implementation that allows running MPI applications across multiple clusters. MPICH-G2 is an integration of the popular MPICH [96] implementation with the Globus Toolkit [86]. The Globus Toolkit is used to stage in/stage out executables and files, start processes on remote resources and combine different communication methods available in a heterogeneous environment (e.g., vendor-specific protocols within clusters with TCP/IP on the inter-cluster links).

Other implementations of MPI which address some grid issues are PACX-MPI [90] which provide grid-aware collective communications or MetaMPI [76] which support multiple communication protocols. MagPIE [115] is a library of MPI-like collective operations optimized for hierarchical, wide-area systems.

- *Performance:* MPI applications typically achieve high performance on cluster supercomputers. Achieving high performance in grid computing requires the programmer to explicitly manage heterogeneity. For example, the programmer has to take various processor speeds into account when distributing work. Also, the communication hierarchy has to be taken into account. MPI provides features that make such optimizations possible. Asynchronous operations can be used for latency hiding. MPICH-G2 uses the communicator construct to deliver the topology of the underlying platform to the programmer.

- *Ease of use:* Message passing is a cumbersome and error-prone programming style compared to semi-automatic parallelization provided by higher-level models, such as grid superscalar or master-worker. The programmer has to explicitly deal with load-balancing and inter-processor communication. As mentioned above, in order to achieve satisfying performance, the programmer also needs to explicitly manage some aspects of the underlying platform, such as communication hierarchy and large differences in processor speeds.

- *Applicability:* The majority of applications can be programmed in message-passing style. MPI is especially suitable for SPMD programs.

- *Fault tolerance, malleability, migration:* There are two approaches to providing fault tolerance, malleability and migration in MPI applications. One approach is providing them *transparently* to the application programmer. This is usually done using system-level checkpointing and/or message logging. This approach was adopted for example in: Co-check MPI [160], Starfish [14] MPI and MPICH-V [49]. A transparent implementation of task migration has been proposed in MPI-TM [152]. AMPI [101] supports malleability and migration via *processor virtualization*: the programmer is presented with a *virtual processor* abstraction and the runtime system dynamically maps virtual processors to physical processors. An advantage of system-level approaches is that little or no effort is required from the application programmer. Disadvantages are complexity, large amount of data that needs to be saved and lack of portability.

  Another approach is to let the programmer provide fault tolerance, malleability or migration. Various extensions and modifications of the MPI standard were proposed. For example, the MPI-2 standard [137] extends the basic MPI standard with primitives for dynamic process management: creating new processes and process termination. FT-MPI [78] proposes extending the set of possible communicator states from valid, invalid to (OK, PROBLEM, FAILED). If a communicator is in an erroneous state, it needs to be rebuilt according to the specified semantics: shrink (shrink the communicator to exclude the failed processors), blank (creates a communicator with 'gaps' that have to be filled before the communicator can be used for communication), rebuild (rebuilds the communicator by starting new processes to fill the 'gaps'). SRS [171] is a library supporting application-level checkpointing for MPI applications. With SRS the programmer has to specify which variables need to be checkpointed and when checkpointing has to take place.

- *Adaptivity:* MPI itself does not provide support for adaptivity. Adding adaptivity to an MPI application is the responsibility of the application programmer. Adaptive MPI applications have been developed in the context of the GrADS project [173]. Each time a performance degradation of the application was detected, the application was checkpointed and restarted on another set of resources. The SRS software has been used to perform the migration. The application programmer needs to supply a *performance model* for the application which allows predicting application runtimes on various set of resources. Also a *resource selector* has to be created which uses the performance model to select a resource set which results in the shortest application runtime. Further, the application needs to be instrumented with sensors that collect application information and detect a performance degradation.

- *Portability:* Grid-enabled MPI implementations hide many platform-specific details which enhances portability. However, MPI is typically used in combination with C or Fortran. Applications written in those languages cannot be

ported to another architecture without recompilations. Java bindings of the MPI interface exist, such as MPJ [55]. However, the message-passing paradigm does not integrate well with object-oriented Java [131]. Communication models based on method invocations, such as Group Method Invocation (GMI) [131] fit better into the Java model.

ProActive

ProActive [30] is a Java middleware which supports the so-called Object-Oriented SPMD programming model [35]. This model is similar to the SPMD model supported by MPI. Whereas an MPI application consists of a number of processes, a ProActive application is structured as a set of *active objects*. Like passive objects, active objects serve incoming method invocations. Additionally, each active object has its own thread of control. Method calls to active objects are *asynchronous* with transparent *future objects*. ProActive provides various group communication primitives based on method invocations.

ProActive provides a convenient deployment mechanism: *deployment descriptors*. The goal is to remove any references to the software and hardware configuration from the application code, so that the application can run unmodified on different configurations. The application has access to *virtual nodes*. An external XML descriptor file specifies the mapping of the virtual nodes to JVMs and the ways the JVMs should be started, for example it specifies the shell command that should be used to start a JVM or a local resource manager to obtain nodes. Starting JVMs can also involve using grid application deployment tools such as the Globus Toolkit.

- *Performance:* ProActive applications can achieve high performance. In [103] a speedup of 100 on 150 nodes has been reported for a parallel solver for 3D Maxwell equations. However, since ProActive is an explicit communication model, the programmer is responsible for applying grid-specific optimizations.
- *Ease of use:* ProActive supports an explicit message passing programming model. The disadvantages of explicit message passing has been already mentioned in the discussion of MPI. However, ProActive is based on Java which is a higher-level programming language than C or Fortran, which are typically used in combination with MPI.
- *Applicability:* Since the programming model supported by ProActive is relatively low-level, a broad variety of applications can be programmed with this programming environment.
- *Fault tolerance, malleability, migration:* ProActive supports migration of active objects between JVMs. The migration is either self-triggered or initiated by an external entity. This facility can be used to implement application malleability and migration. ProActive also provides transparent fault tolerance through Communication Induced Checkpointing.
- *Adaptivity:* Providing adaptivity is the responsibility of the application programmer. No adaptive ProActive applications have been developed to date.

- *Portability:* Portability of ProActive applications is ensured through the use of the Java technology. The deployment descriptors hide the details of the underlying platform from the application enhancing its portability.

Remote Procedure Calls

The concept of Remote Procedure Calls (RPC) [40] has been widely used in programming distributed applications. RPC is similar to message passing, however, instead of sending a message to a remote machine, a routine is called on this machine. With message passing the message has to be explicitly received. With RPC this is not the case. Typically a new thread is created on the receiver to serve the incoming procedure call.

Java's Remote Method Invocation (RMI) [10] is an object-oriented variant of RPC. RMI allows invoking methods on objects located in remote Java Virtual Machines. The suitability of Java RMI for grid computing was investigated in [177]. This research has shown that many high-performance applications can be programmed using Remote Method Invocations and run efficiently in grid environments. The disadvantages of RMI are similar to other explicit message-passing models (such as MPI): the programmer has to explicitly deal with issues like load balancing, communication hierarchy and varying processor speeds. Additional disadvantages of RMI are: lack of asynchronous method calls which makes latency-hiding difficult and lack of group operations.

GridRPC [154] extends RPC with a number of important primitives. Apart from synchronous procedure calls, the GridRPC API defines also *asynchronous* calls and primitives to operate on those calls, e.g. to monitor the status of a previously submitted call, to cancel a call or to wait for *any* of multiple, previously submitted calls. In that way, GridRPC supports fork-join type of parallelism. GridRPC is suitable for medium-to-coarse-grained parallel applications but not for fine-grained parallelism. Example implementations of GridRPC are Netsolve [27] and Ninf [164].

- *Performance:* Applications based on RPCs can achieve high performance in grid environments. For example, in [177] a data-parallel application programmed with RMI has been shown to achieve in wide-area setting performance close to single-cluster performance. However, it is the responsibility of the programmer to apply grid specific optimizations. GridRPC supports this by providing for example asynchronous procedure calls.

- *Ease of use:* Like other explicit communication models, programming with RPCs is difficult since the programmer has to explicitly deal with complex grid programming issues.

- *Applicability:* A broad variety of applications can be programmed with RPCs.

- *Fault tolerance, malleability, migration:* Some RPC frameworks, such as RPC-V [69] provide transparent fault tolerance. With other frameworks, providing fault-tolerance, malleability and migration is the responsibility of the programmer.

- *Adaptivity:* When programming with RPCs, providing adaptivity is the responsibility of the application programmer. However, some implementations of GridRPC API provide a form of transparent adaptivity. For example, Ninf-G uses dynamic information from Network Weather Service [181] to dynamically select the best resource to execute an RPC call.

- *Portability:* The portability of an RPC/RMI application depends on the sequential language used. Using Java enhances the portability of an application.

2.3 Satin: a divide-and-conquer framework

Satin is a framework for writing divide-and-conquer applications developed by Rob van Nieuwpoort [175]. Satin has been inspired by Cilk [46] (hence the name) – a C-based divide-and-conquer framework designed for shared-memory machines. Satin has been designed to run efficiently in grid environments. Satin is Java-based which allows Satin applications to run across heterogeneous grids without the need of recompilation. Programming with Satin is very easy: in order to create a parallel grid application, the programmer annotates the sequential code with divide-and-conquer primitives. The Satin compiler and runtime system take care of the low-level issues, such as inter-processor communication and load balancing. Satin uses a load balancing algorithm called Cluster-aware Random Work Stealing. This algorithm allows Satin applications to achieve high performance in heterogeneous, wide-area environments.

In the remainder of this section, we will describe Satin's programming model and illustrate it with code examples. Next, we will briefly describe Satin's runtime system and the Cluster-aware Random Work Stealing load-balancing algorithm.

2.3.1 The divide-and-conquer paradigm

Divide-and-conquer algorithms operate by dividing the problem at hand into smaller subproblems. The division process continues until the problems become trivial to solve. The solutions of subproblems are combined to provide the solution of the parent problem. A typical example of a divide-and-conquer algorithm is the famous *quicksort* algorithm for sorting arrays of real or integer numbers (Figure 2.2). In the divide phase, a *pivot* element is chosen (thick lines in Figure 2.2) – this can be any element of the array, for example the first one. Next, the array is partitioned into 2 smaller arrays: an array consisting of elements smaller or equal to the pivot element and and array consisting of elements greater than the pivot element. This partitioning is performed in place by swapping elements that are in wrong positions. Then the same procedure is applied to the smaller arrays and is repeated until the size of the arrays reaches 1. In the combine phase arrays are 'glued' together.

Because the subproblems (also called *tasks* or *jobs*) in a divide-and-conquer computation are independent, such a computation can be parallelized by executing different tasks on different machines. Moreover, the task graph of a divide-and-conquer application has a *hierarchical* structure. Therefore, such applications can be executed with good communication locality on hierarchical grids.

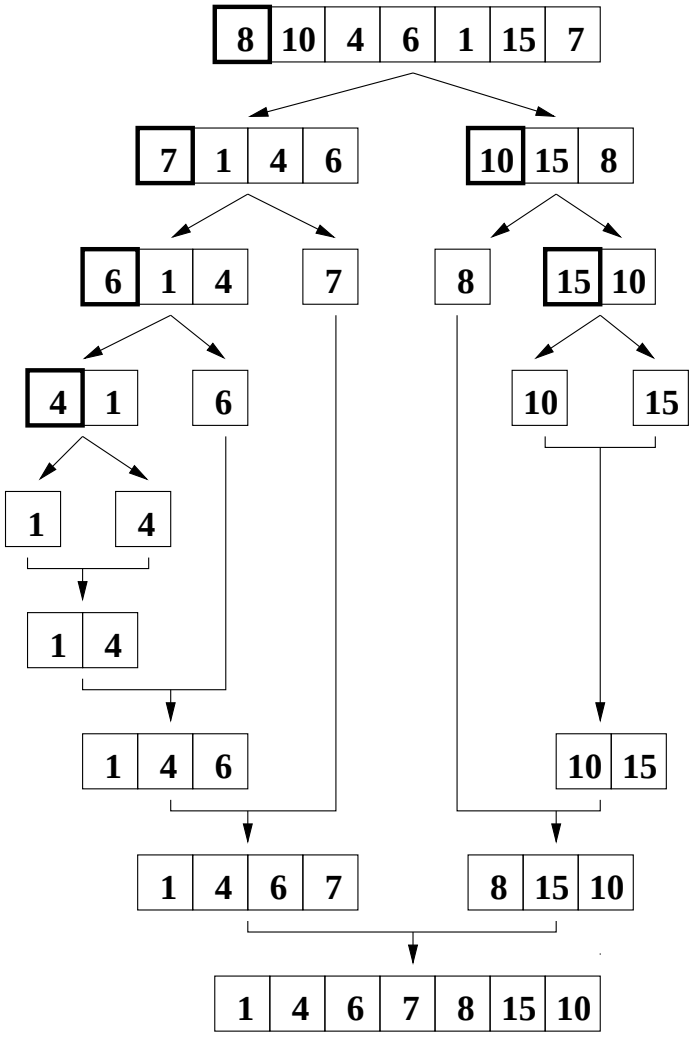


Figure 2.2: The quicksort algorithm

The divide-and-conquer model has many applications. Examples of divide-and-conquer computations include: search and optimization problems (e.g. the satisfiability problem [97]), astrophysical simulations (e.g., the Barnes-Hut N-body algorithm [34]), grammar based learning [12], parallel rendering (raytracing), bioinformatics computations, computational geometry problems (e.g., convex hull calculation), adaptive data classification procedures and numerical methods (e.g multigrid algorithms [184]). Also, all the master-worker computations can be expressed in the divide-and-conquer model. In fact, divide-and-conquer is a *generalization* of the master-worker model: master-worker can be seen as a divide-and-conquer with one level of recursion. The master-worker paradigm has gained extreme popularity in grid community and a vast majority of existing grid applications has been written using this paradigm, for example the famous SETI@home project [7] and similar initiatives [2, 4, 1, 5], the GridSAT satisfiability solver [64], etc. The advantage of divide-and-conquer over master-worker is not only its broader applicability, but it also solves several performance issues. With master-worker computations, the performance of the master process can become a bottleneck of application performance: the speed of the master limits the number of workers that can be used and therefore it limits the speedup that can be achieved. Moreover, master-worker may suffer from communication overhead between the master and workers, especially if they are located on different clusters. This problem can be alleviated by using the hierarchical master-worker paradigm [110]. The hierarchical master-worker grid system uses two levels: a single supervisor process controls multiple master processes. There is one master per site and each master controls a set of workers located on the same site. In this way the amount of wide-area communication is reduced. The divide-and-conquer paradigm can be seen as a further generalization of the hierarchical master-worker paradigm.

2.3.2 The Satin programming model

Satin extends the Java model with two Cilk-like divide-and-conquer primitives: *spawn* and *sync*. While Cilk introduces new keywords into C to implement those primitives, Satin integrates cleanly into Java, without the need of language extensions.

The *spawn* operation is a special form of method invocation. A *spawnable* method can potentially be executed in parallel with the method that has invoked it. We call such an invocation a *spawned method invocation*. The programmer indicates which methods are *spawnable* by means of *marker interfaces* (this mechanism is used in Java RMI). The programmer declares spawnable methods in an interface which extends the special, empty *satin.Spawnable* interface. Each invocation of a method declared in such a way is a spawned method invocation.

Sync is a synchronization operation with the following semantics: wait until all the methods spawned by the current method complete and return their results. Only after the *sync* operation has returned are the results of the spawned methods available. Before *sync*, the values of the variables containing those results are undefined. *Sync* is a method defined in the class *satin.SatinObject*. Each class that spawns work needs to extend the *SatinObject* class and inherits the *sync()* method.

```
 1: interface RaytracerInterface extends satin.Spawnable() {
 2:   BitMap render(Scene scene, int x, int y, int w, int h);
 3: }
 4:
 5: class Raytracer extends satin.SatinObject
 6: implements satin.Spawnable {
 7:
 8:   BitMap render(Scene scene, int x, int y, int w, int h) {
 9:
10:     BitMap picture1, picture2, picture3, picture4;
11:
12:     if (w < THRESHOLD && h < THRESHOLD) {
13:       return renderSequentially(scene, x, y, w, h);
14:     } else {
15:       picture1 = render(scene, x, y, w/2, h/2); /*spawn*/
16:       picture2 = render(scene, x+w/2, y, w/2, h/2); /*spawn*/
17:       picture3 = render(scene, x, y+h/2, w/2, h/2); /*spawn*/
18:       picture4 = render(scene, x+w/2, y+h/2, w/2, h/2); /*spawn*/
19:       sync();
20:       return combinePictures(picture1, picture2, picture3, picture4);
21:     }
22:   }
23:
24: }
```

Figure 2.3: Raytracer: an example divide-and-conquer application in Satin

Figure 2.3 shows an example Satin application: Raytracer: a rendering application that uses the raytracing method. It takes an abstract scene description as an input and outputs a bitmap. The application is parallelized by recursively dividing the picture into four smaller pictures until a certain threshold is reached. Below the threshold the pictures are rendered sequentially. After rendering the smaller pictures the final image is reassembled.

In Figure 2.3 the interface *RaytracerInterface* (line 1) extends the *satin.Spawnable* interface. Therefore, the *render(...)* method (line 2) declared in the *RaytracerInterface* is marked as spawnable. Each invocation of this method (lines 15–18) will be a spawned invocation, which means that *picture1*, *picture2*, *picture3* and *picture4* will be (potentially) rendered in parallel. The *Raytracer* class extends the *satin.SatinObject* class to inherit the *sync()* method and implements the *RaytracerInterface*.

The parameter-passing semantics of spawnable methods are different than the semantics of normal Java methods. Where a spawnable method is executed remotely, the call-by-value semantics are used. However, when a spawnable method is executed locally, the call-by-reference semantics are applied to avoid the overhead of copying the possibly large parameters. Since at the moment a method is spawned it is unknown whether it will be executed remotely or locally, the programmer cannot assume either call-by-value or call-by-reference semantics. Therefore, the programmer must make sure that the application works correctly if either call-by-value or call-by-reference semantics is used.

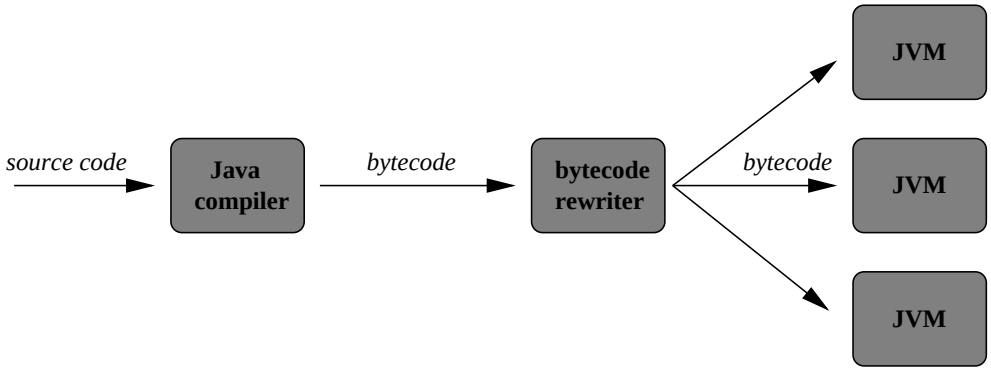


Figure 2.4: Compiling Satin applications

Satin does not provide shared memory. The only way of sharing data between tasks is by explicit parameter passing and returning results. Global variables should not be used by spawnable methods. In other words, spawnable methods should not have side effects. In chapter 5, we will show how this model can be extended with a shared-object abstraction which allows data sharing between independent tasks.

2.3.3 Implementation

The Satin framework consists of a bytecode rewriter and a runtime system. The application code is first compiled with a standard Java compiler (*javac*) and then rewritten by the bytecode rewriter which transforms it into a parallel application (Figure 2.4). The bytecode rewriter replaces each spawned method invocation and each *sync()* operation with a call to the Satin runtime system. For each spawned method invocation the Satin runtime system creates a datastructure called *invocation record*. An invocation record contains the references to the parameters of the method (*not* copies of the parameters; the parameters are copied only if the method is executed remotely) and some extra administration data. The method described by the invocation record is not invoked immediately. Instead, the invocation record is placed in the *work queue* – a datastructure maintained by the runtime system and containing unprocessed *tasks* (spawned method invocations).

For each method that spawns work a *spawn counter* is created - an object that counts the outstanding spawned method invocations. Each time a method is spawned, the spawn counter of its *parent* (the method that invoked it) is increased. Each time a spawned method returns, the spawn counter is decreased.

In the *sync* call, the spawn counter of the current method is checked. If its value is 0, the control is returned to the current method. Otherwise, tasks from the work queue are executed. If the work queue is empty, the Satin runtime system performs load balancing by means of work stealing: it contacts another node, and downloads a task (an invocation record), which it subsequently executes. The choice of a victim

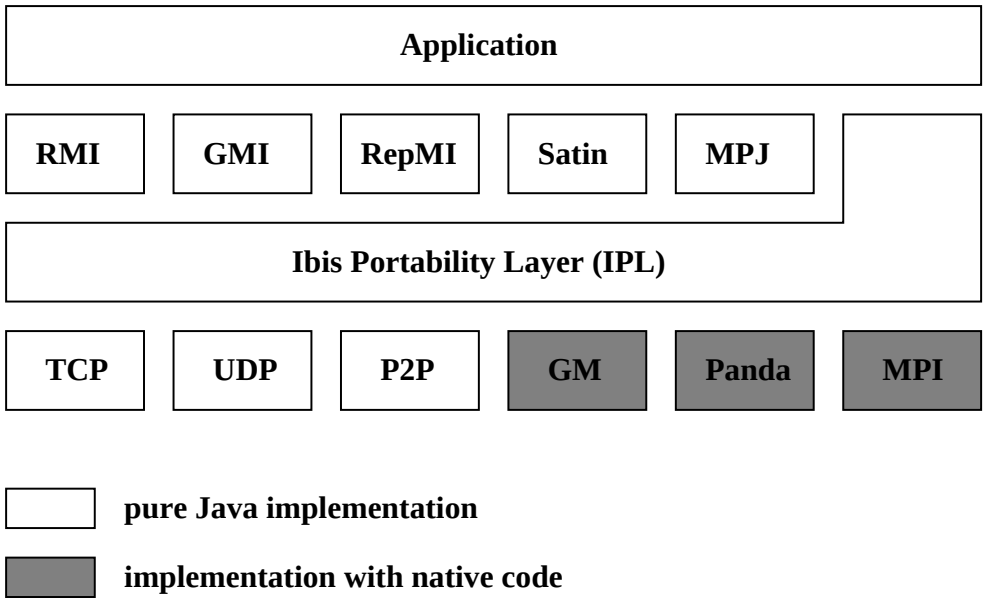


Figure 2.5: The design of Ibis

for work stealing is very important for the application performance. The Satin's work stealing algorithm will be described in more detail in the next section.

When an invocation record is inserted in the work queue, it is put at the head of the queue. In a *sync* operation, if a local task is executed, it is also taken from the head of the queue, so that the queue works as a stack. However, if a task is stolen from a remote node, it is taken from the tail of the remote node's work queue. In divide-and-conquer computation, larger jobs tend to be located towards the tail of the queue and stealing large jobs reduces communication overhead.

The Satin runtime system has been implemented on top of the Ibis communication library [179]. The structure of Ibis is shown in Figure 2.5. The core of Ibis is the Ibis Portability Layer which consists of a number of well-defined interfaces. The application programmer can use the IPL directly or can program with one of the higher-level programming models implemented on top of IPL. Those models include: RMI (remote method invocations), GMI (asynchronous and group communication), RepMI (object replication), Satin and MPJ (MPI-like message passing in Java).

The IPL can have different implementations that can be selected and plugged into the application at runtime. The application needs to specify its communication requirements, such as unreliable/reliable communication, point-to-point/group communication, etc., and the Ibis runtime system selects the appropriate Ibis implementation.

Ibis includes both pure Java implementations based on the TCP, UDP or peer-

to-peer technology and a number of specialized implementations with native code, for example an implementation based on the Panda communication library [39], MPI or GM. The pure Java implementation can be used everywhere, but if an Ibis application is running on a system where Panda, GM or MPI is available, a specialized implementation can be used. Ibis includes a number of optimizations that make the communication more efficient. For example, Ibis offers an optimized object serialization implementation.

Ibis, apart from communication facilities, provides the Ibis Registry. The Registry provides, among others, a membership service to the processors taking part in the computation. The application processes can use this service to discover other processes taking part in the application. The Registry also offers fault detection. Finally, the Registry provides the possibility to send signals to application processes. Currently the Registry is implemented as a centralized server.

2.3.4 Load balancing

Satin balances the load using a work stealing approach. When a processor runs out of work, it steals a task from another processor. The choice of the victim is important for the performance of the application. For homogeneous systems, *Random Stealing* (RS) has been shown to be the optimal strategy [47]. With RS, the victim is chosen *at random*, with uniform probability, from all processors. In grid environments, however, RS performs suboptimally. Because of the uniform probability with which the victim is selected, typically the majority of steal requests are sent to a remote site (cluster/supercomputer). Stealing is done *synchronously*, that is, the thief waits idly until a reply arrives. In grid environments, this means waiting a wide-area round trip most of the times.

Cluster-aware Random Stealing (CRS) [176] is a load-balancing algorithm designed especially for hierarchical systems. CRS distinguishes between nodes in the local site and in remote sites. When a node runs out of work, it first tries to steal from a node in a remote site. However, this wide-area steal request is performed *asynchronously*: the thief does not wait until a reply arrives. Instead, it sets a flag indicating that a wide-area steal is in progress and starts synchronous stealing in the local site. Even if the node finds a job in the local cluster, the wide-area steal request is not canceled. If it is successful, the job is simply put in the work queue. Only one wide-area steal request at a time is allowed – as long as the flag is set, only local stealing will be performed. Victims for both wide-area and local stealing are chosen at random. With wide-area stealing, each node in any remote site has the same probability of being chosen. With local stealing, nodes in the local site are chosen with uniform probability.

Because wide-area stealing is done asynchronously, CRS efficiently hides wide-area latencies. Also, compared to RS, CRS sends much less wide-area messages and thus saves wide-area bandwidth. The performance of CRS was evaluated both in simulations and in a real grid environment – the GridLab testbed. On the GridLab testbed, it achieves 80% efficiency, while the efficiency of RS ranges from 26% (daytime) to 62% (nighttime). Table 2.3.4 contains some information about the nodes used in this experiment. The latencies between the nodes ranged from 1 millisecond to 3.5

location	architecture	Operating System	nodes	CPUs/ node	total CPUs
Vrije Universiteit *Amsterdam* *The Netherlands*	Intel Pentium-III 1 GHz	Red Hat Linux kernel 2.4.18	8	1	8
Vrije Universiteit *Amsterdam* *The Netherlands*	Sun Fire 280R UltraSPARC-III 750 MHz 64bit	Sun Solaris 8	1	2	2
ISUFI/High Perf. Computing Center *Lecce, Italy*	Compaq Alpha 667 MHz 64bit	Compaq Tru64 UNIX V5.1A	1	4	4
Cardiff University *Cardiff, Wales, UK*	Intel Pentium-III 1 GHz	Red Hat Linux 7.1 kernel 2.4.2	1	2	2
Masaryk Univ. *Brno* *Czech Republic*	Intel Xeon 2.4 GHz	Debian Linux kernel 2.4.20	4	2	8
Konrad-Zuse Zentrum für Informationtechnik *Berlin, Germany*	SGI Origin 3000 MIPS R14000 500 MHz	IRIX 6.5	1	16	16

Table 2.1: Nodes used in the GridLab experiment

seconds. The bandwidths ranged from 9 KByte/s to 11 MByte/s. The application used in this experiment was the Raytracer. More details about his experiment can be found in [178].

2.4 Satin vs other GPEs

Satin is an *application development* tool. It does not provide application deployment functionalities. Satin can be combined with any application deployment tool, for example, in our grid experiments we have used Satin in combination with the Globus Toolkit and Zorilla.

Satin provides the programmer with a *high-level* programming model. The application programmer needs only to decompose the problem into tasks that can be done in parallel. The Satin compiler and runtime system take care of the low-level issues such as load balancing and inter-process communication. Below, we will investigate which non-functional properties we have identified in section 2.2.2 are met by Satin.

- *Performance:* Satin achieves excellent performance in grid environments. A Satin application has been shown to achieve parallel efficiency of 80% in a heterogeneous, wide-area environment. Such high performance can be achieved

because of the hierarchical structure of divide-and-conquer applications which suits the structure of grid platforms and the use of the CRS load balancing algorithm. The application programmer does not need to make any special effort to optimize the application for grid environments. The grid-specific optimizations are applied by the compiler and the runtime system.

- *Ease of use:* As a high-level programming model, Satin is extremely easy to use. To create a grid application, the application programmer only needs to annotate the sequential code with the simple divide-and-conquer primitives: *spawn* and *sync*. The runtime system takes care of the low-level issues.

- *Applicability:* A broad range of applications can be expressed in the divide-and-conquer model. This includes all master-worker computations (as divide-and-conquer is a generalization of master-worker), search and optimization problems, astrophysical simulations, parallel rendering etc.

  However, the applicability of the divide-and-conquer paradigm is limited by the lack of global state. The only way of sharing data between tasks is by explicit parameter passing. This model is insufficient for many applications. In chapter 5, we will show how the divide-and-conquer model can be extended with a shared-abstraction: shared objects. This will extend the applicability of our programming model to for example branch-and-bound applications, games with transposition tables, VLSI routing and many others.

- *Fault tolerance, malleability and migration:* In chapter 3, we will show how we can provide *transparent* support for fault-tolerance, malleability and migration. We will present a divide-and-conquer-specific algorithm which allows Satin applications to run on variable numbers of nodes with little overhead.

- *Adaptivity:* Since Satin uses a *dynamic* load-balancing algorithm, it can adapt to varying processor speeds. However, if a difference in processor speeds becomes too large, for example because another, high-priority application overloads part of the processors, the performance might suffer. The overloaded processors will not perform enough computation to amortize the overhead they cause by stealing work from other processors. Also, the prototype Satin implementation could not adapt to changing network conditions. If a certain network link became overloaded and the bandwidth drops beneath a certain threshold, the performance of the application would decrease dramatically. In chapter 4 we will show, how we can make Satin applications adapt to changing conditions in grid environments.

- *Portability:* The portability of Satin is ensured by the use of the Java technology. Thanks to Java's 'write once, run anywhere' property, Satin applications can run unmodified on heterogeneous resources.

  Tables 2.2 and 2.3 provide an overview of all application development tools discussed in section 2.2.2 and a comparison of those systems to the Satin framework. We compare them to both the prototype Satin system implemented by

Rob van Nieuwpoort and to the full system which is the result of the work described in this thesis.

	perf. optimizations applied	ease of use	applications
GRID superscalar	automatically	+	fork/join coarse grained
MW	automatically	+	master-worker
Workflow systems	automatically	+	very coarse grained
HPJava	by programmer	+/-	all, but most suitable for data-parallel
MPI	by programmer	-	all, but most suitable for SPMD applications
ProActive	by programmer	-	all
RPC	by programmer	-	all
Satin (prototype)	automatically	+	divide-and-conquer
Satin (full system)	automatically	+	divide-and-conquer with data sharing

Table 2.2: The comparison of Satin and other grid programming environments

	FT, malleability, migration	adaptivity	portability
GRID superscalar	-	-	+/-
MW	+	+/-	-
Workflow systems	+	only some	varies
ch HPJava	-	-	+
MPI	only some implementations	-	-
ProActive	+	-	+
RPC	-	-	+ (RMI) - (others)
Satin (prototype)	-	-	+
Satin (full system)	+	+	+

Table 2.3: The comparison of Satin and other grid programming environments

Chapter 3

Fault tolerance, malleability and migration

3.1 Introduction

In grid environments, the availability of computing resources changes constantly. Processor crashes are more likely to occur than in traditional parallel environments. Also, since there is no centralized control, computing nodes may be rebooted or shut down for maintenance with or without prior notice. Finally, processors may be taken away from the application because they are claimed by another, higher-priority application, because a processor reservation has ended. On the other hand, new processors might become available.

A grid application must be able to adapt to such changes in order to survive in a grid environment and achieve good performance. In this chapter, we will discuss three issues that are important for grid applications to adapt to changes in grid environments:

- *fault tolerance* – the ability of an application to operate in the presence of hardware and software failures, i.e. processors and network crashes.
- *malleability* – the ability of an application to handle processors joining and leaving an on-going computation.
- *migratability* – the ability of an application to transfer to a different set of computational resources during the run.

The three above issues are closely related to each other. For example, if an application can handle crashing processors (fault tolerance) and continue working on the diminished number of processors, it can also handle leaving processors (partial malleability). However, if the processors are leaving *gracefully* (i.e., after a prior notice) handling it may be more efficient than handling crashing processors. Further, if an application is malleable, it is also migratable: it can be migrated from one set of

resources to another by first adding the new processors to the computation and then removing the old ones.

In this chapter, we will present a novel technique to provide fault tolerance, malleability and migratability to divide-and-conquer applications. We will describe its implementation in Satin and evaluate its performance.

The rest of this chapter is structured as follows. Section 3.2 contains background information on fault tolerance, malleability and migration. In section 3.3, we will present our fault-tolerance algorithm. In section 3.4, we will describe how the fault-tolerance algorithm can be extended to handle malleability. In section 3.5, we will further extend our fault-tolerance algorithm to handle total crashes. In section 3.6, we will evaluate the performance of our algorithms. In section 3.7, we compare our approach with related work. Finally, we conclude in section 3.8.

3.2 Background

In this section, we will discuss some background information on fault tolerance, malleability and migration issues.

3.2.1 Failure models

To achieve fault tolerance in a distributed system or application, it is important to know the *failure model* of the system components. A failure model characterizes the behavior of a component in case of a failure. The literature lists a vast number of failure models with various degrees of 'severity'. A failure model is *more severe* than another failure model if the set of faulty behaviors allowed by it is a superset of the set of behaviors allowed by the other model [138]. The most commonly used models are *crash failure* and *arbitrary failure* also known as *Byzantine failure*. *Crash failure* is the least severe failure model. In this model, a faulty process stops prematurely but it was working correctly before it stopped. *Byzantine failure* is the most severe failure model and it states that a faulty process might exhibit any behavior whatsoever. Most fault-tolerance techniques, including the one presented in this chapter, assume the crash failure model. There are also techniques known that can deal with Byzantine failures. The techniques for handling both crash and Byzantine failures will be described briefly hereafter.

3.2.2 Fault-tolerance techniques

In this section, we will describe the most important approaches to implementing fault tolerance in distributed applications. We will cover checkpointing, message logging, retry (recomputing) and replication.

Checkpointing

The most popular fault-tolerance mechanism is *checkpointing*, i.e., periodically saving the state of the application on *stable storage*, a device that can survive failures

– usually one or more hard disks. The information stored on the stable storage is called a *checkpoint*. After a crash, the application is restarted from the last checkpoint rather than from the beginning [165]. Checkpointing comes in three varieties: *uncoordinated checkpointing*, *coordinated checkpointing* and *communication induced checkpointing* [77].

With uncoordinated checkpointing, each process takes its checkpoints independently. This allows to avoid the synchronization overhead. Finding a consistent set of checkpoints to roll back to might be difficult, however. Rolling back a crashed process may cause rolling back other, dependent processes that have sent or received messages from the crashed process. This *rollback propagation* might extend back to the initial state of the computation (*domino effect*) [149].

The domino effect can be avoided by using *coordinated checkpointing* or *communication induced checkpointing*. With coordinated checkpointing, the processes synchronize before taking a checkpoint to make sure that the resulting set of checkpoints is consistent. The disadvantage of coordinated checkpointing over uncoordinated checkpointing is the synchronization overhead. The advantage is that the recovery is faster and easier to implement.

With communication induced checkpointing, processes take two kinds of checkpoints: local and forced. Local checkpoints are taken independently by each process. Forced checkpoints are taken if a message exchanged by two processes could cause creation of a *useless* checkpoint, that is, a checkpoint that will never be a part of a consistent global state [77]. This guarantees that the domino effect will not occur.

In practice, the most commonly used technique is coordinated checkpointing [77]. The reason is that, currently, the main cause of overhead is access to stable storage and not synchronization. The simplicity of the recovery procedure is also an important argument.

Checkpointing can be done either at the system level or at the application level. With system-level checkpointing, the system-level state of the application is saved. The advantage of system-level checkpointing is that it is completely *transparent* to the application programmer. However, the system-level implementation of checkpointing can be extremely complex, as has been shown in the Dynamite project [106]. Not only do the memory image, stack and registers of a process need to be saved, but also its signal mask, open file descriptors and open network connections. Reproducing the open file descriptors after a process has been restarted from a checkpoint is nontrivial, because the files might not be accessible on the machine where the process is restarted. Restoring network connections requires complex protocols. Finally, system-level checkpointing is inherently not portable, since process checkpoints contain OS-specific data, and a process checkpointed under one OS cannot be restarted on another OS.

With application-level checkpointing, the application itself saves its critical variables and datastructures. Application-level checkpointing is typically easier to implement. It often requires the cooperation of the application programmer, however, and is therefore not *transparent*. Further, application-level checkpointing is more portable than system-level checkpointing, as the checkpoint data does not contain OS-dependent information. Finally, application-level checkpointing is more efficient

since smaller amounts of data need to be saved.

Checkpointing is used in grid computing by such systems as Condor [169], Dynamite [106] (system-level checkpointing), Cactus [22] (application-level checkpointing) and the European DataGrid project [92] (application-level checkpointing). Also, several MPI implementations provide checkpointing facilities, for example CoCheck MPI [160], Starfish MPI [14] and MPICH-V [49].

The main advantage of checkpointing is that it is a very general technique which can be applied to any type of parallel applications. The disadvantage is that it causes execution time overhead, even if there are no crashes. This overhead depends on the frequency with which checkpoints are taken and the programmer must be careful in choosing a reasonable frequency. In [185] and [174], formulas are presented which can be used to calculate the optimal checkpointing frequency. However, the programmer needs to have a detailed knowledge about the characteristics of the application and the system it is running on, such as the time it takes to save a checkpoint and the mean-time-to-failure.

The overhead of checkpointing can be reduced using such techniques as *concurrent checkpointing* [145] and *incremental checkpointing* [80]. With concurrent checkpointing, the execution of a process is continued while its state is being saved to stable storage. Incremental checkpointing avoids rewriting the portions of the process state that have not changed since the previous checkpoint.

Another problem of most checkpointing schemes is the complexity of the crash recovery procedure, especially in dynamic and heterogeneous grid environments where rescheduling the application and retrieving and transferring the checkpoint data between nodes is non-trivial. The final problem of checkpointing is that in most existing implementations, the application needs to be restarted on the same number of processors as used before the crash, so it does not support malleability. An exception is SRS [171], a checkpointing library for MPI applications which saves data in such a way that an application can be restarted on a different number of processors.

Message Logging

An alternative fault-tolerance technique is *message logging*: during failure-free operation, each process logs sent or received messages (depending on the variant of message logging algorithm) from other processes [77]. After a failure, the crashed process is re-executed and the logged messages are replayed. Message logging protocols assume a *piecewise deterministic model*: the execution of each process is deterministic between occurrences of non-deterministic events. The non-deterministic events are usually receipts of messages, but the protocol can be easily extended to handle other types of non-deterministic events. All non-deterministic events need to be logged.

Message logging is typically combined with checkpointing to reduce the amount of re-execution needed – message logging enables the system to recover beyond the last checkpoint [77]. Therefore, message logging is also often used to provide the applications the ability to interact with the outside world. Message logging is used less often than checkpointing. An example of a system that uses a combination of message logging and checkpointing is MPICH-GF [183] or MPICH-V [49]. Message

logging can also be combined with other fault-tolerance techniques. For example, RPC-V [69] combines message logging with replication.

Message logging schemes come in three flavours: *pessimistic message logging*, *optimistic message logging* and *causal message logging*. Pessimistic message logging does not allow any message to be received before it is logged. This approach guarantees that so-called *orphan processes* are never created. An orphan process is a process that depends on a message that has not been logged and whose sender has crashed. The disadvantage of this approach is a high performance overhead. Logging messages affects communication throughput and latency. The advantage of pessimistic logging is the simplicity of the recovery procedure: processes other than the crashed process are not affected by the crash.

Optimistic logging tries to reduce the logging overhead by making the optimistic assumption that logging will complete before a crash occurs [77]. Messages are logged asynchronously so a message can be received before it is logged. This reduces the logging overhead but significantly complicates the recovery procedure. Optimistic logging does not exclude the creation of orphan processes. Such processes must be rolled back during the recovery procedure.

Causal message logging also avoids synchronous access to stable storage while avoiding creating orphan processes at the same time. Causal logging ensures that each message on which a process causally depends (according to Lamport's *happened-before* relation [118]) is either logged or available locally (in the volatile memory) to that process. This is implemented by piggybacking messages in the process' memory which have not been logged on each message the process sends to another process. The recovery procedure with causal logging is more complex than in case of pessimistic logging. In practice, pessimistic logging is most commonly used because of the simplicity of the recovery procedure [77].

The advantages and disadvantages of message logging techniques are similar to those of checkpointing techniques. Message logging is a very general technique but it can cause high execution time overhead. It can affect communication throughput and latency. With some message-logging protocols, if stable storage is accessed through the network, the bandwidth required by the application doubles. Also, message logging cannot be used to implement malleability: the application cannot continue execution on the diminished number of processors, the crashed processor needs to be replaced.

Replication

Replication is another approach to implementing fault tolerance. Multiple copies of the same task/process are run on separate processors. If one of the copies crashes, other copies are used. This technique can be used not only for tolerating crash failures but also Byzantine failures. In the latter case, replication is combined with voting: the result returned by the *majority* of replicas is considered valid, other results are discarded. To tolerate N crash failures, N+1 replicas are needed. To tolerate N Byzantine failures, 3N+1 replicas are needed. This technique is suitable for systems of which high-availability is required, since the recovery is fast – it basically requires

switching to another replica.

Replication is often used in hardware-based fault tolerance. An example is Triple Modular Redundancy used in electronic systems.

An example of software-based fault tolerance using the replication principle is the mechanism used in the FTAG runtime system [61]. The FTAG programming language is based on the functional paradigm. A FTAG program is structured as a set of modules. Modules can be decomposed into sub-modules, which resembles the divide-and-conquer style programming. With FTAG, the user can select one of the two supported failure models: crash failure or Byzantine failure. The computation is replicated for fault-tolerance purposes. The replicas exchange partial results. If the crash failure model is selected, this exchange of partial results is used to speed up the computation: if a replica receives a result of a certain sub-module and it does not need to compute this sub-module anymore. If the Byzantine failure model is used, majority voting is used for each partial result to determine its correctness.

Another example of a system that uses software-based replication is RPC-V [69]. RPC-V combines replication with message logging.

Retry

Another technique used for providing fault tolerance is *retry – recomputing* parts of the work that were lost in a crash. This technique cannot be applied to an arbitrary application. One group of applications to which this technique can be applied are applications structured as a series of (possibly nested) *atomic actions* [129]. In case of a processor crash, an atomic action can be aborted without side-effects and restarted from the beginning.

Applications that adhere to the functional programming paradigm can also use this principle [108]. Functional programming applications consist of functions with no side-effects. There is no notion of global state and the result of a function depends only on its input parameters. Function execution will always produce the same outputs if given the same inputs, a property known as *referential transparency* [61]. So, in case of a crash, functions executed by crashed processors can be re-executed.

One example of applications that adhere to the functional programming paradigm are master-worker applications. Master-worker tasks are typically functions whose results depend solely on their parameters and with no side-effects. Fault tolerance in master-worker applications is typically implemented by recomputing tasks done on crashed workers. A separate fault-tolerance technique needs to be applied to the master – usually checkpointing or replication. An example of a master-worker framework that adopts this fault-tolerance mechanism is MW [95] (see also section 2.2.2). Charlotte [33] introduces a fault-tolerance mechanism called *eager scheduling*. It reschedules a task to idle processors as long as the task's result has not been returned. Crashes can be handled without the need of detecting them. Assigning a single task to multiple processors also guarantees that a slow processor will not slow down the progress of the whole application.

Divide-and-conquer applications also adhere to the functional paradigm and therefore the retry principle can be used for providing fault tolerance in this type of appli-

cations. However, this naive approach might lead to large amounts of recomputation when a task located high in the hierarchical task graph is lost in a crash. Also, naive recomputation might cause the need of recomputing work done by processors that have not crashed. In this chapter, we will explain in more detail why the naive recomputing approach is not adequate for divide-and-conquer applications and we will present a more efficient solution. Other divide-and-conquer frameworks which use recomputing to achieve fault tolerance are: Cilk [46], CilkNow [44], Atlas [32], DIB [83] and Lin and Keller's work [126]. A more detailed description of the algorithms used by those systems and their comparison to the algorithm described in this chapter will be given in the related work section at the end of the chapter.

3.2.3 Malleability techniques

The basic idea behind implementing transparent malleability in parallel applications is separating parallelizing, that is, identifying what can be done in parallel, from mapping to physical processors [101]. For SPMD (MPI-like) applications, this can be done by *processor virtualization*. The programmer operates on virtual processors, the number of which is typically many times bigger than the number of physical processors. The runtime system takes care of mapping the virtual processors to the physical one. Malleability can be achieved in two ways. One way is migrating virtual processors off leaving or to joining physical processors. Another way is checkpointing the application in such a way that each virtual process has a separate checkpoint file. The application can then be stopped, checkpoint files rearranged and the application restarted on a different number of processors. This approach is used in Adaptive MPI [101] (virtual processor migration and checkpointing) and Phoenix (only checkpointing).

Another approach is to treat the number of processors the application is running on as a variable. The data partitioning depends on the value of this variable. When this value is fixed at the time the job starts and cannot be changed during the run, we call the application *moldable* [111]. Many data-parallel and SPMD applications are written in that way. Moldable applications can be turned into malleable applications by introducing *reconfiguration points* at which the number of processors can be changed. This approach is used in DyRecT [93], DRMS [9] and SRS [171]. At a reconfiguration point, global synchronization and data redistribution takes place. Data redistribution can be done by means of group communication (DyRecT, DRMS) or checkpointing (SRS).

Master-worker and divide-and-conquer paradigms are especially attractive when implementing malleability. When programming with those paradigms the programmer does not use the notion of processors. Instead the notion of *tasks* or *jobs* is used. The tasks are mapped to the physical processors by the compiler or runtime system. Joining processors are handled in a straightforward manner by assigning tasks from the pool of free tasks to those processors. Leaving processors can be handled using the fault-tolerance mechanism: leaving processors are treated as crashing processors. Some systems, however, can handle gracefully leaving processors (i.e., after a prior notification) more efficiently than processor crashes. For example, Piranha [56] allows the programmer to specify a 'cleanup' procedure which is called when a task

needs to vacate a leaving processor. In this thesis, we will also present a malleability mechanism that is an 'optimized' version of the fault-tolerance mechanism.

3.2.4 Migration techniques

In sequential applications, migration is traditionally achieved by stopping the application execution on the current node, transferring the whole application state to the new node and restarting the application on the new node from the point where it was stopped on the old node. Migration can also be implemented on top of checkpointing: a checkpoint file is created on the old node and transferred to the new node where the application is restarted from the checkpoint file rather than from the beginning. Those two approaches are very similar. In fact, direct migration can be seen as an optimized version of checkpoint-based migration: the data is transferred directly into the memory of the new machine instead of via stable storage [160].

Similarly to checkpointing, migration can be implemented either on the operating system level (system-level migration) or in the application itself (application-level migration). As explained in section 3.2.2, system-level implementations are extremely complex. Care needs to be taken to properly save and restore open file descriptors and open network connections [106]. Also, system-level implementations are not portable. However, implementing migration on the OS level is *transparent* and therefore more convenient for the programmer. Application-level techniques are less complex to implement and more portable. Typically, they are also more efficient, since less data needs to be saved and transferred. However, application-level techniques are not transparent.

Parallel applications can be migrated using the same approach: each process is migrated separately by direct transfer of the process state or by checkpointing. Special care needs to be taken to guarantee that the states of all migrated processes are consistent and that the communication channels between processes are correctly restored after migration. Migration of MPI applications was studied in the Dynamite project [106] also in [101], [152] and [167].

Another approach to migrating parallel applications is using malleability to achieve migration. An application can be migrated from one set of resources to another by first adding the new set of resources to the computation and then removing the old set.

3.3 Fault-tolerance for Satin

The divide-and-conquer paradigm is well suited for implementing fault-tolerance, malleability and migration. There is no notion of global state in a divide-and-conquer application: function execution does not have side-effects and the result of a function depends only on its input parameters. Function execution will always produce the same outputs if given the same inputs, a property known as *referential transparency*. So, the work lost in a crash of a processor can be redone at any time during execution of the application.

Therefore, it is possible to handle leaving or crashing processors by *recomputing* work done by those processors. Such a mechanism has low overhead, as no synchronization between processors is needed and no data needs to be stored on stable storage. Several such techniques have been proposed [32, 44, 83, 126]. However, the common problem of those techniques is *redundant computation* which degrades their performance. They do not reuse *orphan* work, that is, tasks that are dependent on tasks done by leaving processors. Orphan work is discarded and recomputed.

In this section, we will describe a recovery mechanism which salvages orphan work and thus avoids redundant computations. Orphan work is salvaged by restructuring the execution tree. The overhead of our mechanism during crash-free execution is very small. Our mechanism can handle crashes of multiple processors or entire clusters.

In the following sections, we will discuss two simple extensions to the fault-tolerance mechanism. First, we extend the orphan saving scheme in such a way that we can also reuse partial results computed by the *gracefully* leaving processors. This occurs, for example, when the processor reservation is coming to an end or when the application receives a notification that it should vacate part of its processors for another, higher-priority application. When the processors leave gracefully, the work done by them is randomly distributed over the other processors. Then, the orphan saving scheme is used to reuse those partial results. When processors are leaving gracefully, our mechanism can save nearly all the work done by the leaving processors. That, combined with the fact that adding processors to ongoing divide-and-conquer computations is straightforward (they just start stealing), results in efficient *malleability*. We can also use our technique for efficient *migration* of the computation: to migrate the computation from one cluster to another, we first add the new cluster to the computation and then (gracefully) remove the old one.

The disadvantage of this scheme is that always at least one processor must be running, or else all work will be lost. This makes it impossible to stop an application and restart it later from the point where it was stopped. It is also impossible to survive total crashes, i.e. the situations when all processors have crashed. Therefore, we extended the basic scheme with the possibility of storing partial results in a user-defined file. The results stored in the file can be reused using the orphan-saving mechanism.

The resulting system can handle a vast variety of scenarios typical for the Grid:

- Crashing processors, including a total crash can be handled.
- Processors joining and leaving an on-going computation can be handled with high efficiency.
- An application can be efficiently migrated.
- An application can be stopped and restarted later on a possibly different set of resources.

In the remainder of this section, we will describe the basic fault-tolerance mechanism. The extensions will be described in the following sections.

3.3.1 Failure detection

We use two different mechanisms to detect processor crashes. One mechanism is implemented in the communication layer (Ibis). If a connection between two hosts is broken, the communication layer notifies the Satin runtime system. The second mechanism is implemented in the Ibis Registry. The Registry periodically sends a keep-alive message to every node. If a node does not respond to this message within the specified timeout, the Registry notifies the remaining nodes that this node has died.

In general, it is impossible to detect failures reliably in asynchronous systems where message propagation time is unbounded. Therefore, both of our failure detection methods assume that there exist an upper bound on message propagation time. This may result in false positives in some cases. Also, the system cannot distinguish between a crashed processor and a broken network connection. This may also result in false positives. False positives, however, affect the *performance* of our failure recovery algorithm, as some jobs might be recomputed unnecessarily, but not its *correctness*. The system will continue to work correctly as long as the following condition is satisfied:

> If processor A thinks that processor B has crashed, then either processor B has indeed crashed or processor B thinks that processor A has crashed.

We make sure that this condition always holds by breaking all connections with processors that we assume to be crashed.

3.3.2 Recomputing jobs stolen by leaving processors

To be able to recompute jobs stolen by leaving processors, we keep track of all the jobs stolen in the system. Each processor maintains an *outstandingJobs* list containing the *invocation records* of jobs stolen from this processor (invocation records are datastructures describing the jobs, see section 2.3.3). For each job, the processorID of the thief is stored. When one or more processors are leaving or crashing, each of the remaining processors traverses its *outstandingJobs* list and searches for jobs stolen by the leaving processors. If such a job is found, it is put back in the work queue of the processor from which the job was stolen. Later, this job will be recomputed by the local processor or stolen by another processor. Figure 3.1 (a) shows an example computation tree. Four processors are taking part in the computation. Processors store the information about stolen jobs in their *outstandingJobs* queues: processor 1 remembers that job 2 was stolen by processor 3 and job 14 by processor 2. Processor 3 remembers that job 4 was stolen by processor 4. Processors also remember where the jobs were stolen from: this information is stored in the invocation record of each stolen job. Figure 3.1 (b) shows the situation after the crash of processor 3. As soon as processors 1, 2 and 4 discover the crash of processor 3, they search through their *outstandingJobs* lists. Processor 1 discovers that job 2 has been stolen by processor 3 and puts this job back in its work queue (figure 3.2 (a)). Each job reinserted into

a work queue during the recovery procedure is marked as 'restarted'. Children of 'restarted' jobs are also marked as 'restarted' when they are spawned.

3.3.3 Orphan jobs

Orphan jobs are jobs stolen *from* leaving processors. In figure 3.2 (a), job 4 and all its subjobs are orphans. In most existing approaches, the processor which has finished working on an orphan job must discard the result of this job: since the processor where the job was stolen from has crashed, the result cannot be sent back. Orphan jobs are recomputed when their restarted parents are recomputed. For example, in figure 3.2 (a), job 4 and all its subjobs would be recomputed while recomputing job 2. This is undesirable, since a crash of a small number of processors can cause recomputation of large parts of the work, if the crashing processor was computing jobs high in the tree.

The results of orphan jobs are valid partial results and can be used while recomputing their parents. The results of orphan jobs would be usable if the processors recomputing the parents knew where to retrieve those orphans or the orphan task knew the new address to return the result. Thus, salvaging orphan jobs requires creating the link between the orphan and its restarted parent.

We restore links between parents and orphans in the following way: for each *finished* orphan job (jobs 9 and 17 in figure 3.2 (a)), we forward to the other processors a small message containing the jobID of the orphan and the processorID of the processor computing this orphan.

We abort the unfinished intermediate nodes of orphan subtrees, since they require little computation: in a typical divide-and-conquer application, the bulk of the computation is done in the leaf nodes, the intermediate nodes only split work and combine the results. Aborting simplifies the algorithm and eliminates the possibility of deadlocks in Satin. In section 3.3.7, we will discuss an alternative orphan saving scheme in which the unfinished orphans are not aborted. We will show that this makes the algorithm much more complicated and does not improve the performance.

The (jobID, processorID) tuples are stored by each processor in a local *orphan table*. Figure 3.2 (b) shows the computation tree after the recovery procedure. Processor 4 aborted jobs 4, 8 and 16 and forwarded the (jobID, processorID) tuples for jobs 9 and 17 to the other processors. Processors 1 and 2 stored those tuples in their orphan tables. The crash recovery procedure is completed. Note that the crash recovery does not require inter-process synchronization: each processor processes the crashes independently of the other processors.

Jobs that have been restarted after a crash and all their subjobs have a 'restarted' flag set in their invocation records. Before starting the execution of such jobs, processors perform lookups in their local orphan tables. If the jobID of the spawned job corresponds with the jobID of one of the orphans in the table, the processor does not start computing the job. Instead, it puts the job on its *outstandingJobs* list and sends a message to the owner of the orphan requesting the result of the job. Figure 3.3 (a) shows the continuation of the computation from figures 3.1 – 3.2. In the meantime, processor 2 stole job 2 from processor 1 and started executing it. Because it is a

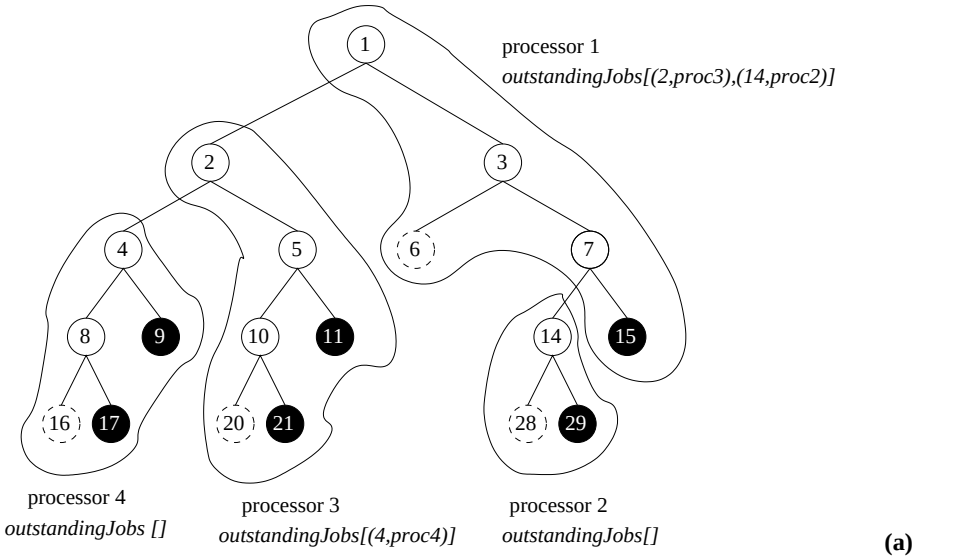
'restarted' job, processor 2 performs a lookup in its orphan table for this job and all its subjobs. After spawning job 9, it discovers that it has an entry for this job in its orphan table. Instead of computing this job, it puts it on the *outstandingJobs* list and *asynchronously* sends a message (*result request*) to processor 4 requesting the result of job 9 (figure 3.3 (a)). Note, that at this moment, the state of the execution tree and the datastructures (*outstandingJobs* lists) is exactly as if job 9 was stolen by processor 4 from processor 2. This has important consequences. First, the result returned by processor 4 can be handled using the normal routine used for handling the results of stolen jobs. Processor 2 does not need to wait until the result is returned. Instead, it can compute other jobs in the meantime. Second, if processor 4 crashes before it returns the result, this crash will be handled by the normal crash recovery procedure: job 9 will be taken from the *outstandingJobs* list and put back in the work queue of processor 2. This guarantees that job 9 will always be computed and that processor 2 will not hang waiting indefinitely for the reply of processor 4.

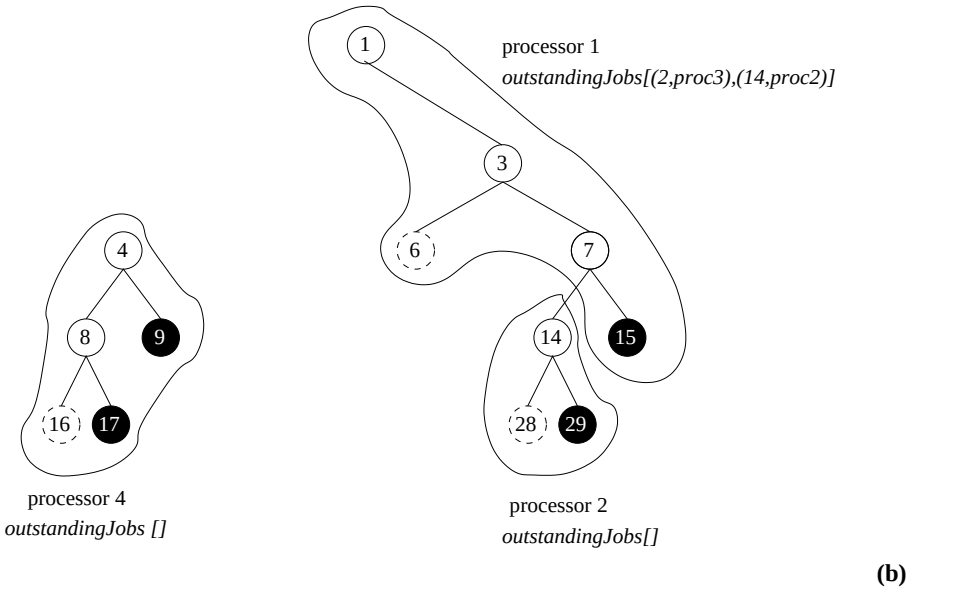
Processor 4, after receiving the result request sends the result of job 9 to processor 2 (figure 3.4 (a)) The format of the message containing this result is exactly the same as a format of a message returning the results of a stolen job. The results of job 17 will be reused in the same way later in the computation.

Note that reusing orphans does not influence the correctness of the algorithm. If the result of an orphan is not found (e.g. because the (jobID, processorID) tuple does not arrive in time), the job can always be recomputed. Reusing orphans is an *optimization* that improves the performance of the system but does not influence the correctness of the crash recovery procedure. This has important consequences for the implementation of the forwarding of the tuples: no reliable and potentially high-overhead broadcast protocols are needed. Currently, we are using asynchronous broadcasting. An alternative solution would be *piggybacking* tuples on other messages sent by the Satin runtime system, for example steal requests and replies. Also, we use *message combining*: instead of sending each tuple in a separate message, we combine multiple tuples into one message. This reduces the number of messages sent during the recovery procedure to one broadcast message per processor.

3.3.4 Orphan propagation

An orphan subtree might not necessarily be located on a single processor like in the example above where the whole subtree of job 4 was located on processor 3 (figure 3.1 (a)). If one of the subjobs of job 4 was stolen, the orphan subtree would be distributed over two processors. For example, in figure 3.5 (a), processor 5 stole job 8 from processor 4. After the crash of processor 2, job 8 and all its subtree become orphans because their ancestor, job 4, was stolen from a crashed processor. However, processor 5 does not have enough information to discover that. Therefore, we introduce *orphan propagation* messages. When a processor discovers that a part of the orphan subtree was stolen by another processor, it sends an *orphan propagation* message containing the identifier of the stolen job to the other processor. Orphan propagation messages are sent asynchronously. Orphan propagation continues recursively, if necessary. In our example, processor 4 sends an orphan propagation message to processor 5 (figure 3.6



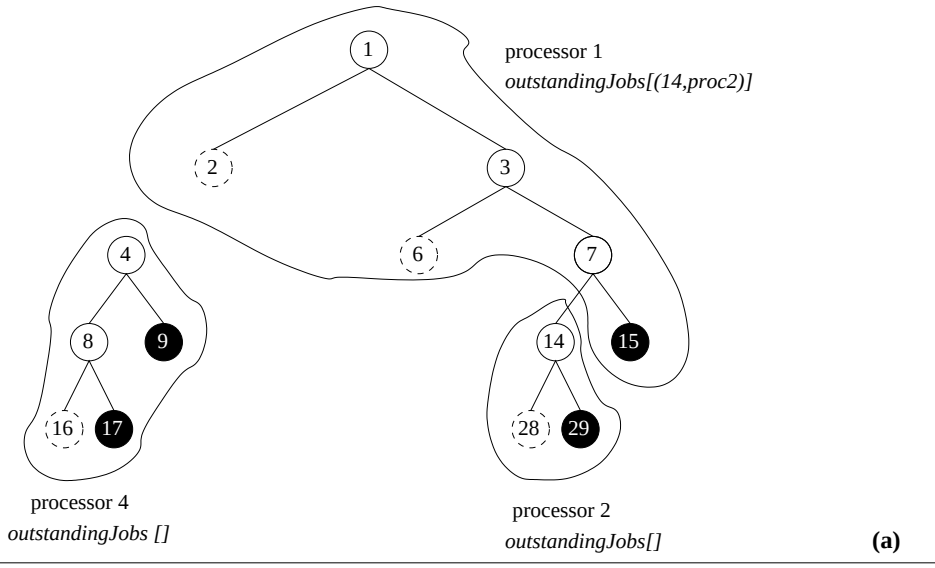


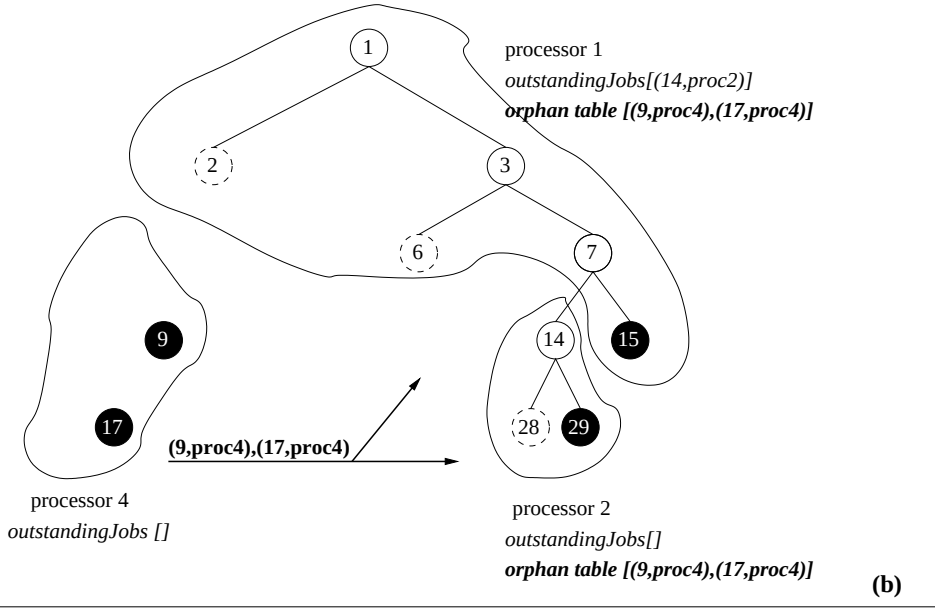
1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.1: An example computation tree before and after the crash of processor 3



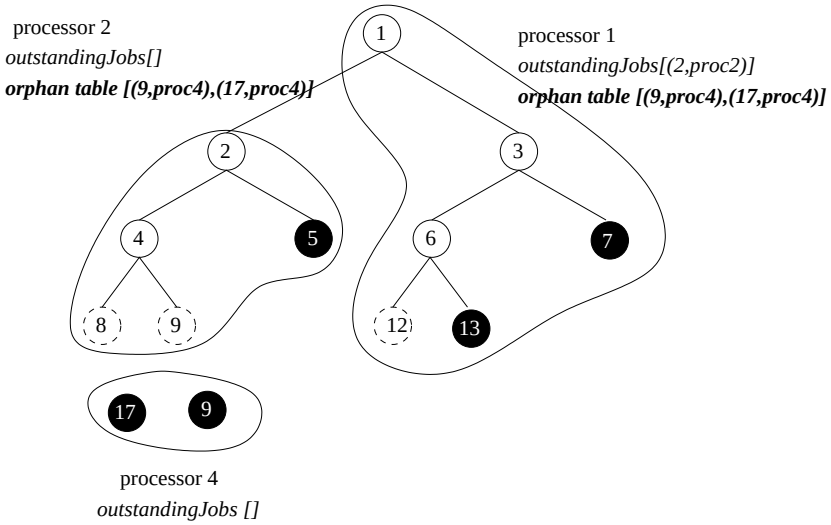


1 Job finished

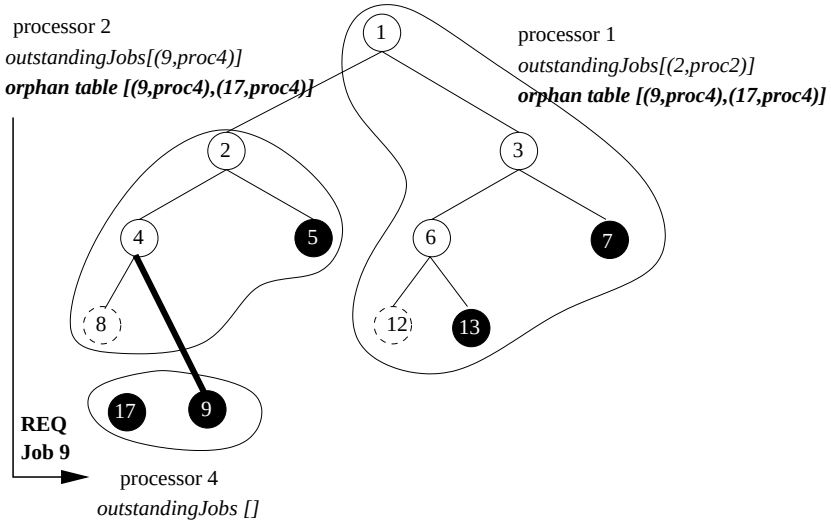
1 Job in progress

1 Job spawned but not yet started

Figure 3.2: The crash handling procedure



(a)



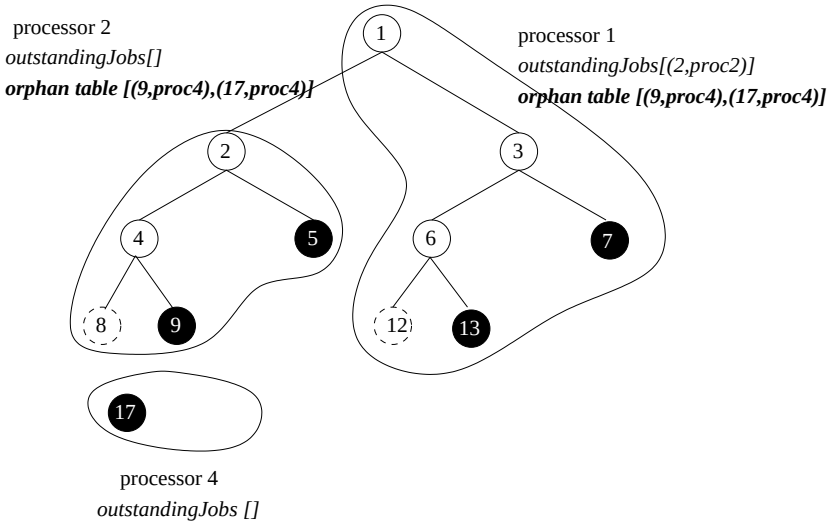
(b)

1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.3: Restoring the parent-child link



1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.4: Processor 4 returns the result of the orphan to processor 2

(a)). Processor 5 aborts jobs 8 and 16 and forwards a (jobID, processorID) tuple for job 17 (figure 3.6 (a)).

3.3.5 Handling crashes of the master processor

The processor that spawned the job that is the root of the execution tree is called the *master*. In figure 3.1 (a), job 1 is the root of the execution tree and processor 1 is the master. A crash of the master is a special case. Since the root job was never stolen, it will not be restarted during the normal recovery procedure in which jobs stolen by crashed processors are restarted. Therefore, a special procedure for handling a crash of the master is needed.

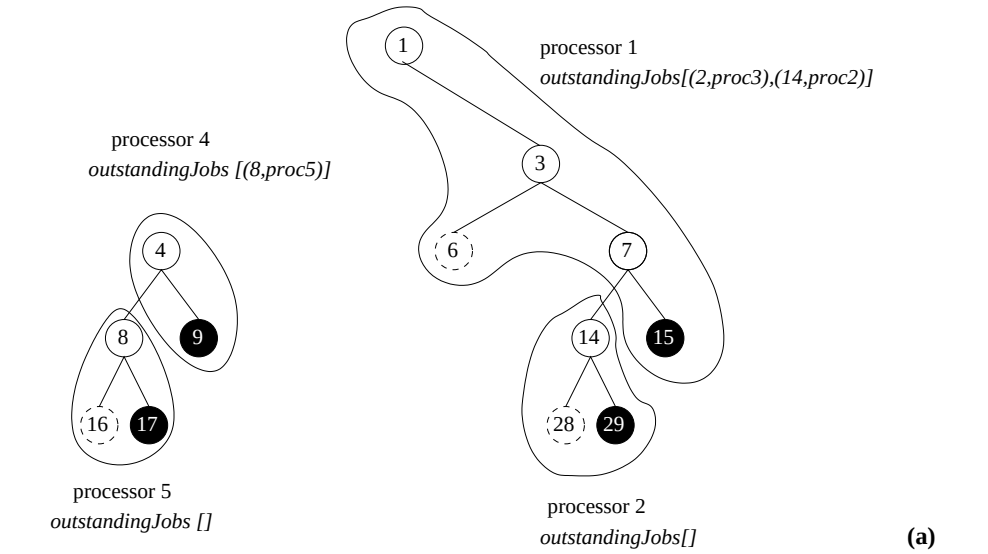
When the crash of the master is discovered, the remaining processors elect the new master using the Registry[1]. The new master re-spawns the root job, thereby restarting the application. The information needed to restart the application is replicated on all processors. The new run of the application will reuse the partial results of the orphan jobs from the previous run (when the master crashes, all jobs become orphans). Figures 3.7 (a) shows the computation tree from figure 3.1 (a) after the crash of the master (processor 1). Figure 3.7 (b) shows the situation after the crash handling procedure. Processor 3 has been elected as a new master and restarted the root of the computation tree (job 1).

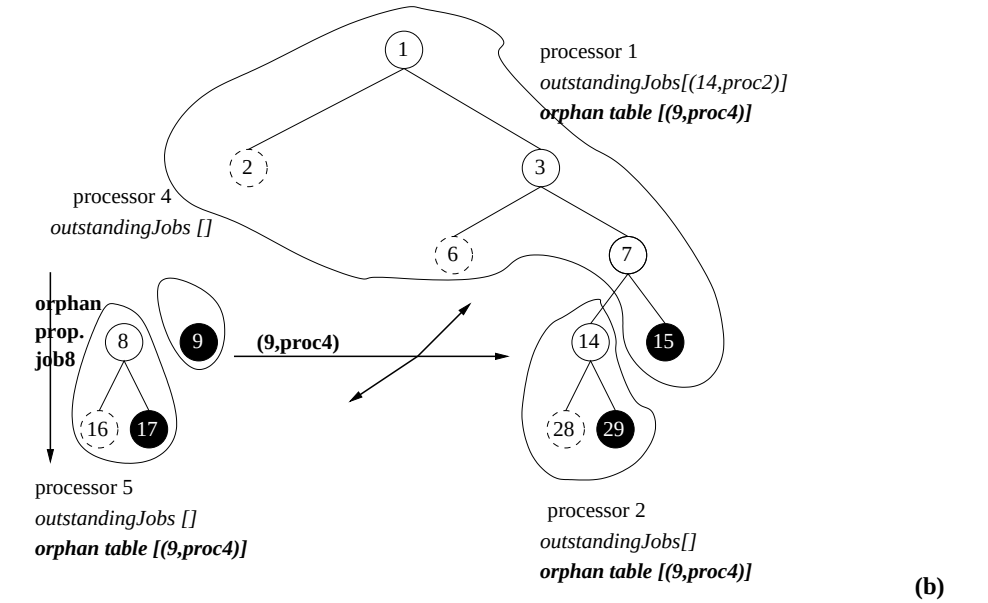
3.3.6 Job identifiers

The job identifiers (jobID) must be both globally unique and *reproducible*: the identifier of a job that is re-spawned after a processor crash must be the same as it was before the crash, otherwise the orphaned children cannot be linked correctly to their parents. We create job identifiers in the following way: the root job is assigned ID=1. The child's identifier is computed by multiplying the identifier of its parent by the maximal branching factor of the computation tree and adding the number of children the same parent generated before. For example, the second child of a job with ID 4 in a tree with branching factor = 2 will have ID = 2 * 4 + 1 = 9. The jobs in the tree in figures 3.1–3.7 are numbered according to this scheme.

In most divide-and-conquer applications, the maximal branching factor of the execution tree is known. If it is not known, however, *level stamps* described in [126] can be used. A level stamp is a string. The root job is identified by an empty string. The level stamp of a child is created by appending a character to the identifier of the parent. The appended character is the number of children the parent has spawned before. For example, the first child of the root job will be identified with a stamp '0', the second child of job '021' will be identified with '0211'. Figure 3.8 shows an example execution tree with level stamps. In our implementation, the application programmer can specify the maximal branching factor of the application. In that case the integer job identifiers are used, otherwise the runtime system uses level stamps.

[1] Crashes of the Registry have to handled by a separate mechanism such as checkpointing and replication



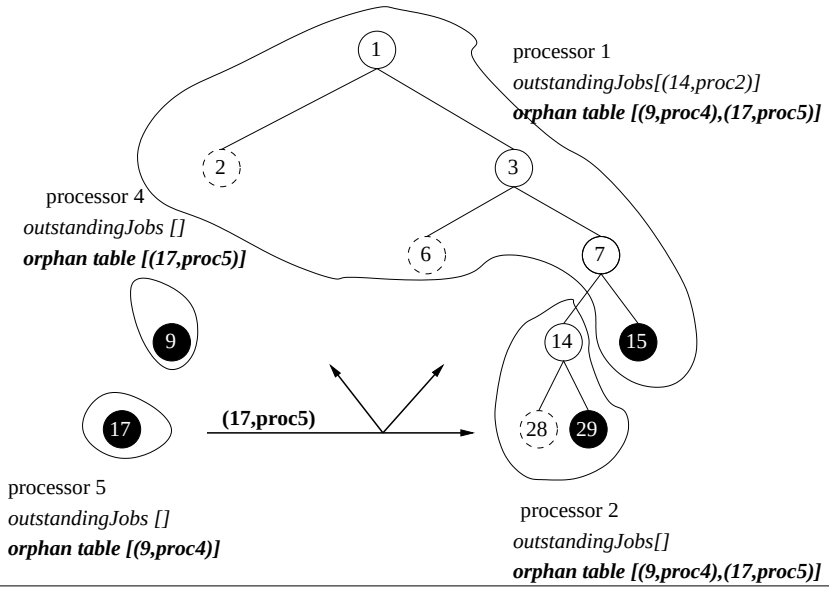


1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.5: Orphan propagation

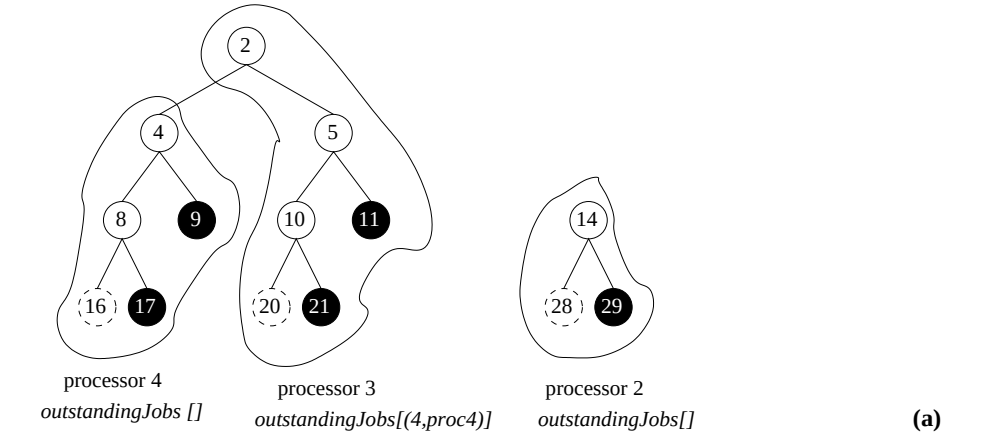


1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.6: Orphan propagation



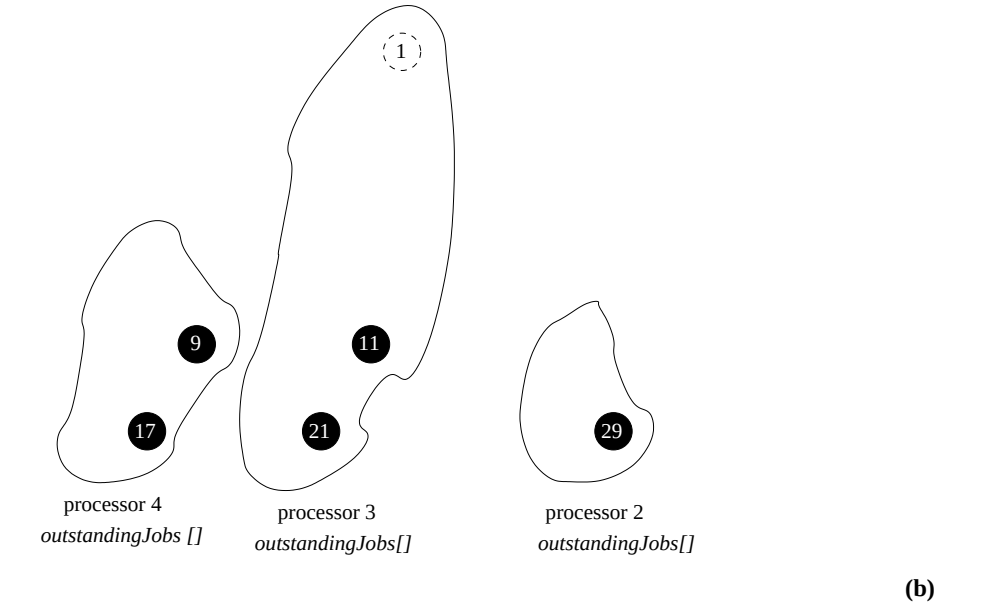


Figure 3.7: Handling the crash of the master (processor 1)

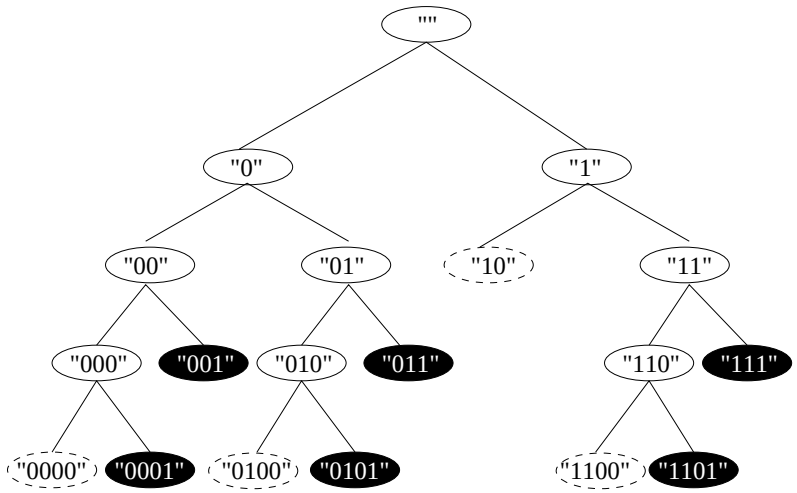


Figure 3.8: Level stamps

3.3.7 Alternative orphan saving schemes

In this section, we will discuss alternative orphan saving schemes and we will explain why they were found to be less efficient.

Global result table

One alternative scheme we tried is using a *global result table* – a concept similar to a transposition table [50] used in game solving environments or the table used in tabled execution of logic programs [163]. It is a table accessible to all processors in which results of jobs can be stored. Jobs in the table are identified by their *parameters*. The global result table is used for storing the results of orphan jobs. As in the basic scheme, only *finished* orphans are stored in the table. Unfinished orphans are aborted. When recomputing jobs lost in crashes, processors perform lookups in the global result table. If a lookup is successful the result found in the table is used instead of recomputing the job.

The global result table is replicated on all processors. The replicas of the table do not have to be strongly consistent. If a processor does not find a job, it can always recompute it. Therefore, updates of the table are propagated to other processors asynchronously.

The global result table scheme has many similarities with the basic scheme. In fact, the basic scheme can be seen as a *distributed* implementation of the global result table: instead of replicating the job results on all processors, the results are stored locally and only *pointers* to the results ((jobID, processorID) tuples) are forwarded to other processors.

The advantage of the global result table scheme over the basic scheme is that

orphan results are always available locally and the result request messages do not need to be sent. This simplifies the algorithm and reduces the number of messages that are sent in the system. However, a severe disadvantage of the global result table scheme is that for applications with large job parameters and large job results, much data is transferred. The problem of large parameters can be solved by using job identifiers described in section 3.3.6 instead of parameters to identify jobs in the global result table. However, there still remains the problem of large results. Therefore, the global result table scheme is not suitable for applications with large parameters and results.

Avoiding aborting orphans

To avoid aborting orphans, we extended the basic orphan saving scheme in the following way. The (jobID, processorID) are broadcast for all orphans, including the unfinished ones. No orphan is aborted. This means that a result request may arrive while the requested orphan job is still not finished. In that case, the information about the processor requesting this job is stored in this job's invocation record: the *owner* field is set to the identifier of the processor requesting the job. For regular jobs (i.e. not orphans) the *owner* field contains the identifier of the processor from which the job was stolen and where the result should be returned. Thus, after the orphan job is finished, its result will be returned to the processor that requested this job as if this job was stolen from this processor.

Unfortunately, this solution introduces a possibility of deadlocks in the Satin runtime system. For efficiency Satin is single-threaded and has one stack. Therefore unfinished orphan jobs can be blocked by their parents which after being restarted can be higher in the stack than their orphaned children. An example of such a situation is shown in figure 3.9. In this figure, the stacks of three processors are shown. Job 2 was restarted after a crash. Jobs 4, 8, 16 and 32 are orphans. The arrows denote parent-child relationships. Job 2 cannot be completed before job 4, because job 4 is its child. Job 4 cannot be completed before job 16, because job 16 is its grandchild. Job 16 cannot be completed before job 2 because it is lower in the stack. In the basic orphan saving scheme such deadlocks are impossible – we reuse only the finished parts of orphan jobs so their execution cannot be blocked by the restarted jobs.

Such deadlock can be avoided by delaying the restarting of the jobs lost in a crash until a *safe moment*. A job can be safely restarted when its *parent* is on the top of the stack. Therefore, after a crash we do not put restarted jobs immediately in the work queue, but store them in a separate queue. A job from this queue is put in the work queue only if its parent is on the top of the stack and the work queue is empty. In this way, we make sure that no orphans will be blocked by their parents. Unfortunately, this approach increases the load balancing overhead of the application. The reason is that putting jobs aside temporarily decreases the number of jobs available in the system. Those jobs are typically relatively large jobs, because restarted jobs have been stolen before, and stolen jobs tend to be large. Thus, the decrease in the number of available jobs can be significant. The performance gain of

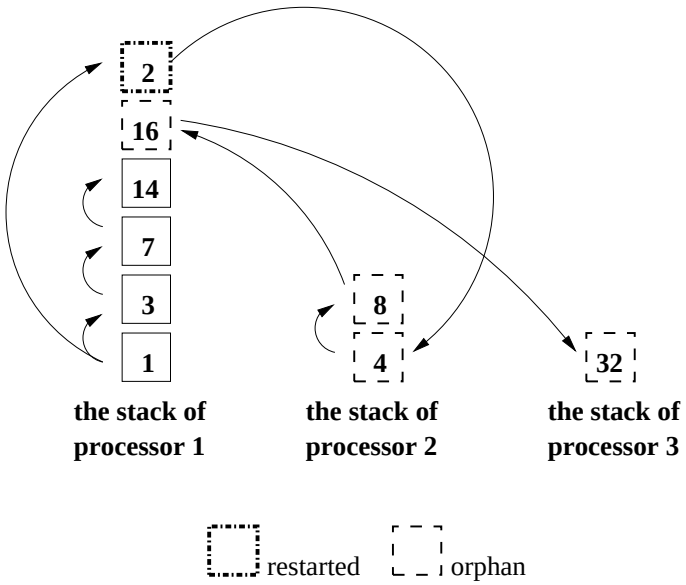


Figure 3.9: An example of a deadlock

not aborting orphans does not outweigh the extra load balancing overhead. Moreover, not aborting orphans makes the algorithm significantly more complicated, increasing the probability of bugs and race conditions.

3.4 Malleability and migration for Satin

An important characteristic of the fault-tolerance algorithm described in the previous section is that after a crash, the application can continue running on the diminished number of processors. The crashed processors do not need to be replaced. Therefore, the applications using our fault-tolerance algorithm are already partly malleable: they can tolerate processors *leaving* the on-going computation. In this section, we will discuss how we can handle processors *joining* the on-going computation. Furthermore, we will show how the crash handling mechanism can be optimized if the application receives a *prior notification* before the processors are taken away. With the optimized mechanism, we can save almost all work done by the leaving processors, reducing the overhead to nearly zero.

3.4.1 Adding processors

Adding a processor to an on-going divide-and-conquer computation is simple. All we need to do is to let the new processor steal jobs from the other processors and the load will be balanced automatically. Adding processors has practically no overhead.

Special care needs to be taken when a processor joins the computation *after* the recovery procedure was executed by other processors (e.g., if new processors were added to replace leaving processors). In this case, the orphan table of the new processor is empty and it has to download an orphan table from one of the other processors, to be able to reuse partial results. The problem here is that a joining processor does not know: a) whether there was a crash recovery before it joined b) which other processors have non-empty orphan tables and which do not (because they have also just joined). We solve this problem in the following way. *Every* processor joining the computation, even processors joining at the very beginning of the computation, *except for* the master, tries to download an orphan table from another processor. Only the master assumes that it has an up-to-date version of the orphan table (it is empty at the beginning of the computation). Each processor *piggybacks* orphan table requests on its steal requests until it receives the table.

3.4.2 Saving partial results from the leaving processors

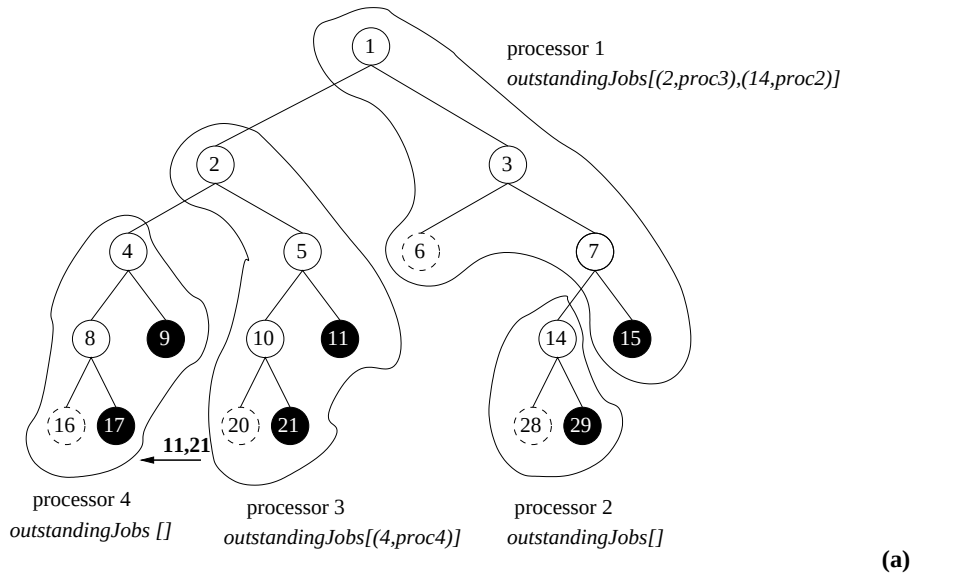
We extended the crash handling algorithm in such a way that if processors are leaving *gracefully*, that is if the application receives a notification before the processors leave, we can save the partial results from the leaving processors.

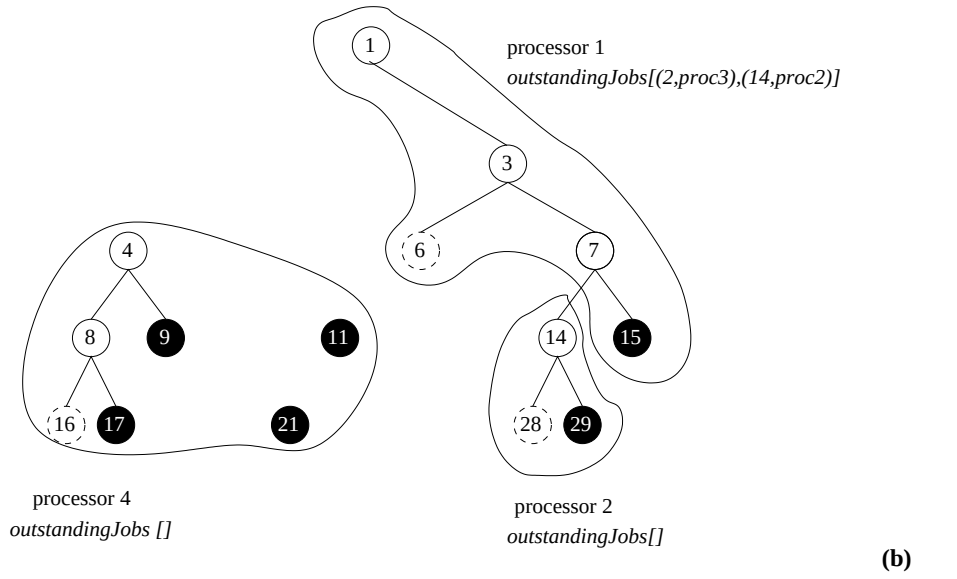
We assume that such departure notifications will be sent to the application by the grid scheduler or other grid middleware. Currently, however, none of the grid schedulers support this functionality. For performance evaluation purposes, we implemented a simple control interface in the Ibis Registry. The user can send a command to the Registry containing a list of nodes that have to leave the computation. The Registry passes this list to all the nodes taking part in the computation.

Our algorithm can also work with other models of departure notification, for example, if notifications are sent only to the leaving processors and if they do not contain the identifiers of other leaving processors. However, in such cases, our algorithm can be less efficient, as will be explained below.

If a processor receives a departure notification, it chooses another processor randomly, transfers all the results of its *finished* jobs to the other processor and exits. The processor that receives those jobs treats them as orphan jobs: it broadcasts a (jobID, processorID) tuple containing *its own* processorID for each received result. Next, the normal crash recovery procedure is executed by all the processors that did not leave. The processors that left are treated as crashed processors. The partial results from the crashed processors are linked to the restarted parents, as it happens in the case of orphan jobs.

An example is shown in figures 3.10 – 3.11. Processor 3 receives a signal that it has to leave the computation. It chooses another processor at random (processor 4) and it sends it all its *finished* jobs – jobs 11 and 21. It aborts the unfinished jobs, and exits. Next, the remaining processors execute the normal crash handling procedure: processor 1 restarts job 2 stolen by processor 3. Processor 4 handles its orphan jobs. Jobs 11 and 21 received from processor 3 are handled in exactly the same was as orphan jobs: for each of the a (jobID, processorID) tuple is sent and stored in the orphan tables. Therefore, the jobs computed by processor 3 can be reused in further



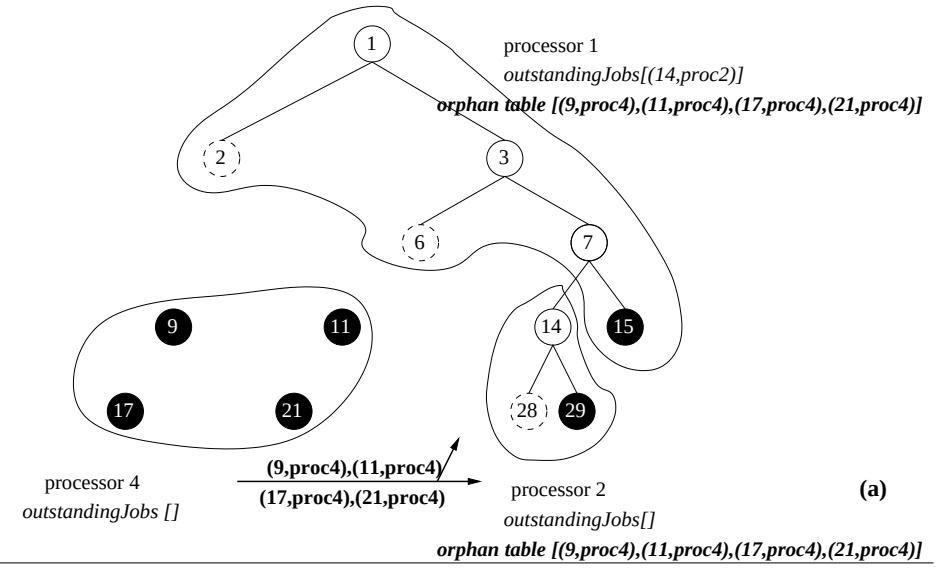


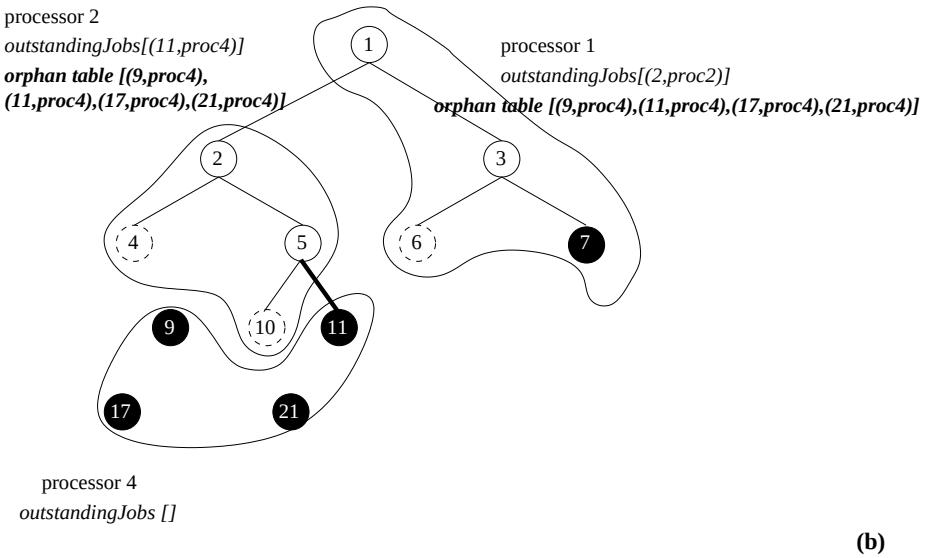
1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.10: Handling gracefully leaving processors





1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.11: Handling gracefully leaving processors

computation (figure 3.11).

The choice of the processor to which the leaving processor will transfer its partial results depends on the information the leaving processor has about the system. Currently, the departure notification received from the Registry contains the identifiers of all processors that are leaving at the same time. Thus, the leaving processors make sure that they transfer their results to one of the processors that are not leaving. However, if the leaving processors do not have full information, it may happen that partial results are transfered to a processor that is leaving as well. If it leaves while the results are in transfer, they will be lost. Otherwise the processor will forward them together with its own partial results to another processor. Note that this only influences the performance of the algorithm and not the correctness: if the results are lost they can always be recomputed.

3.4.3 Using malleability to implement migration

In section 3.6, we will show that our algorithm allows adding and removing processors practically without loss overhead. Therefore, we can use malleability to implement efficient application migration. We can migrate an application from one set of resources to another, by first adding the new set of resources and then removing the old one. Note that order is important – there must be some processors up and running at all times to preserve work.

3.5 Total crashes

A disadvantage of our fault-tolerance and malleability mechanism is that if a processor crashes suddenly, the work done by it is always lost. If a substantial part of the processors crash, a substantial part of work needs to be recomputed. If all processors crash, everything needs to be recomputed. Only if a prior notification is sent to the application, can the work done on the leaving processors be saved. However, if *all* processors are leaving, their work cannot be saved even if a prior notification is sent. Thus, with the current fault-tolerance/malleability mechanism, it is not possible to stop an application and restart it later from the point where it was stopped. The application can only make progress if at every moment there is at least one processor up and running.

To overcome this limitation, we extended our fault-tolerance mechanism to (periodically) store partial results on a stable storage. All processes taking part in the application (periodically) save the results of their *finished* subjobs in a user-defined file.

This mechanism can be used in two ways:

- To minimize the amount of work lost in crashes. In this scenario all processors periodically save their partial results on the stable storage. After a crash, the results computed by the crashed processor are retrieved and reused.
- To stop an application and restart it later from the point where it was stopped.

> In this scenario, the user (or grid middleware) sends a signal to the application, for example via the Registry. After receiving the signal, processes store their results in the file defined by the user and exit. The user can use the file later to restart the application on a possibly different set of resources.

This mechanism can be seen as an application-level checkpointing. The difference with classical application-level checkpointing schemes is that it is done transparently, that is, it does not need to be described explicitly by the programmer. This is possible, because we concentrate on a single class of divide-and-conquer applications. In further text we will refer to our mechanism as *checkpointing*.

3.5.1 The basic checkpointing algorithm

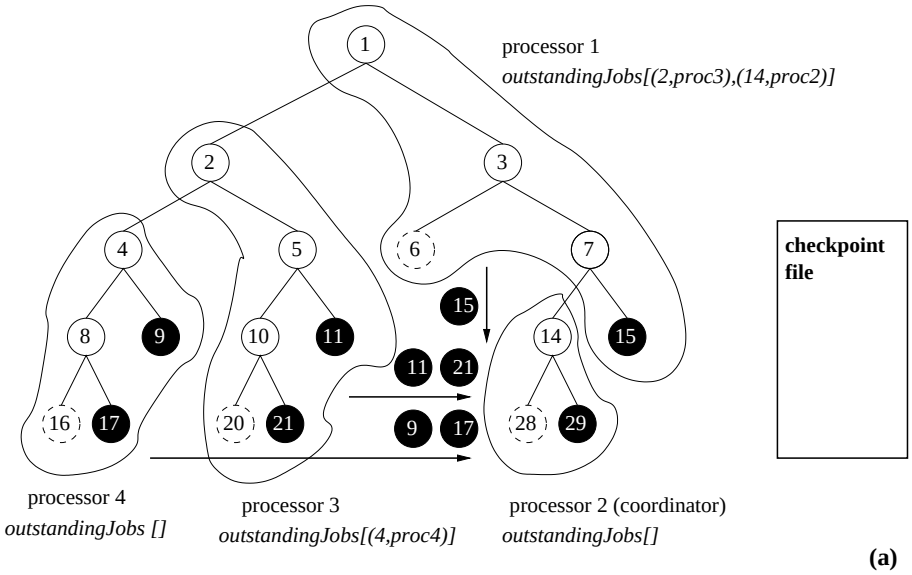
All processors taking part in the computation periodically save their partial results in a user-defined *checkpoint file*. Along with the job results, the jobID of this job and the processorID of the processor that has computed this job are stored. The interval between the subsequent checkpoints (*checkpointing interval*) is defined by the user.

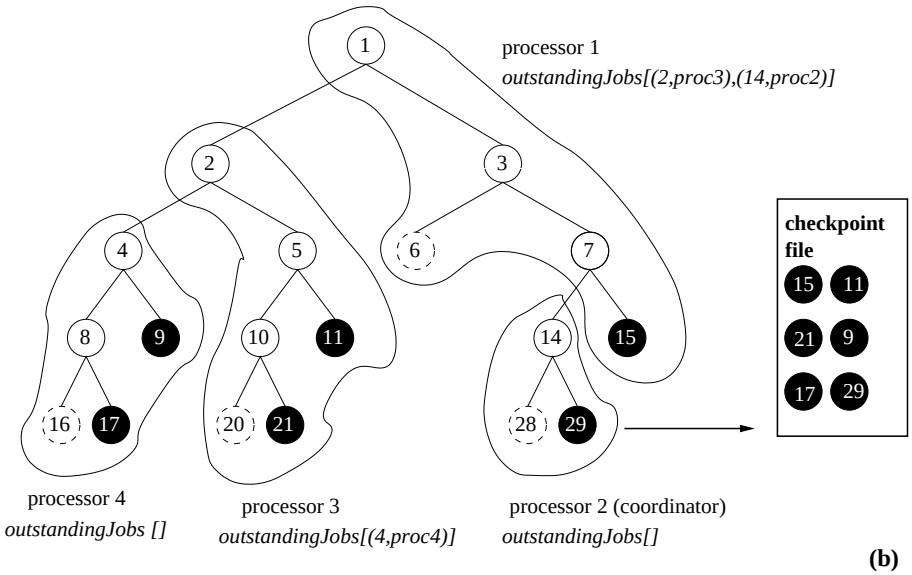
Processors do not access the checkpoint file directly. Instead, they send the data to the *coordinator* processor which is responsible for writing and reading the checkpoint file. The processors *do not* synchronize before taking their checkpoints – the checkpoints can be taken independently. The coordinator is elected from among the processes taking part in the computation. The election algorithm will be described in section 3.5.4. If a processor crashes, the coordinator searches the checkpoint file for the results computed by the crashed processor. All those results are retrieved and stored in the memory of the coordinator. Next, the basic fault tolerance mechanism is used to reuse those results – they are treated just like orphan jobs. For each of those results, the coordinator forwards a (jobID, processorID) tuple with *its own* processorID to the other processors. Processors store the (jobID, processorID) tuples in their orphan tables. The orphan tables are used in exactly the same way as in the basic fault tolerance mechanism.

An example is shown in figures 3.12 – 3.14. All processors periodically send results of their finished jobs to the coordinator – processor 2. The coordinator stores those results together with its own results in the checkpoint file. After the crash of processor 3, a normal crash handling procedure is executed: processor 1 puts job 2 back in its work queue and processor 4 handles its orphans. Additionally, processor 2 searches the checkpoint file for the results computed by processor 3. It retrieves jobs 11 and 21, stores them in its memory and broadcasts the (jobID, processorID) tuples. The tuples are stored in orphan tables and used to reuse the checkpointed results.

3.5.2 Restoring the computation after an abort or total crash

The main advantage of storing partial results on stable storage is the possibility of stopping the computation and restarting it later without the need of recomputing from scratch. Also, surviving a total crash is possible.



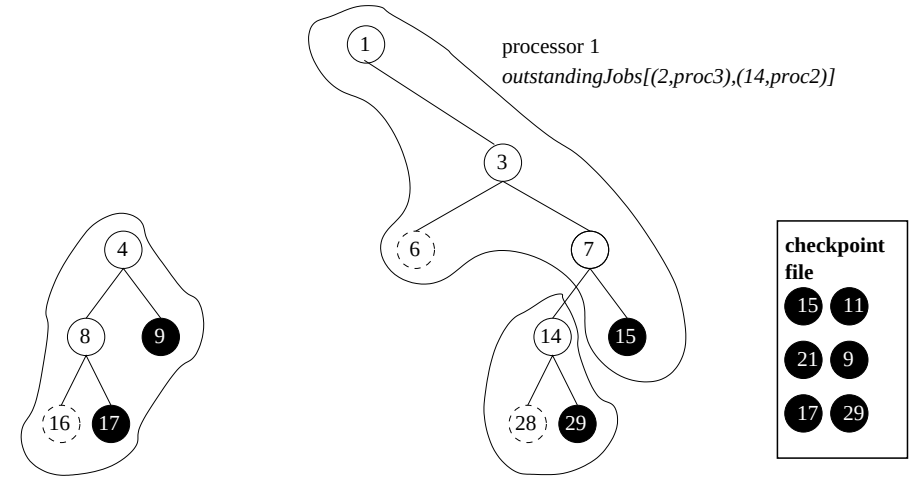


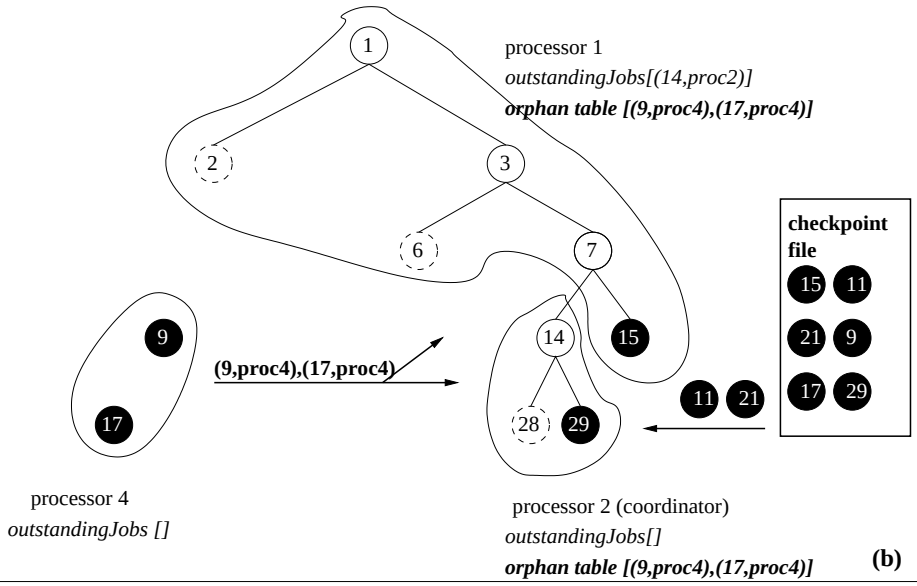
1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.12: Processors are taking a checkpoint



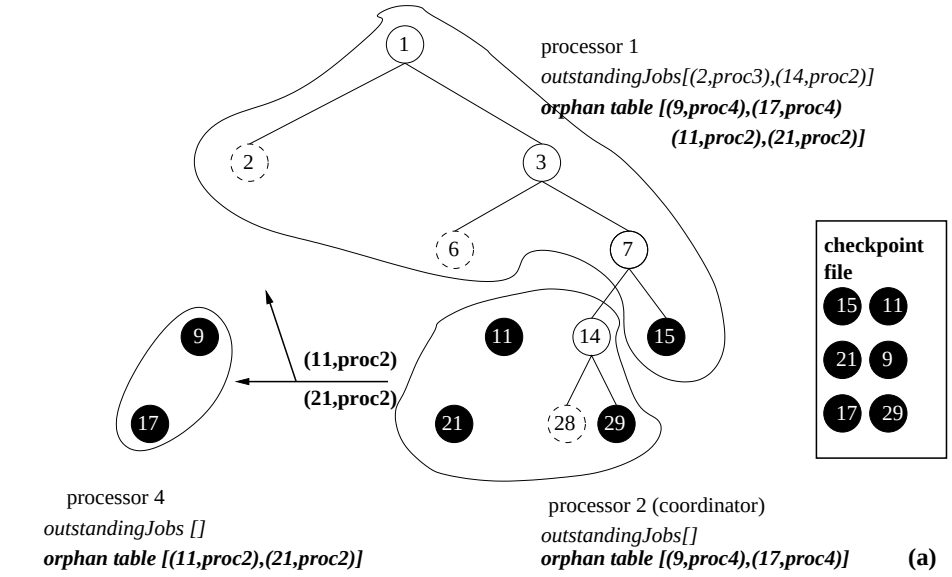


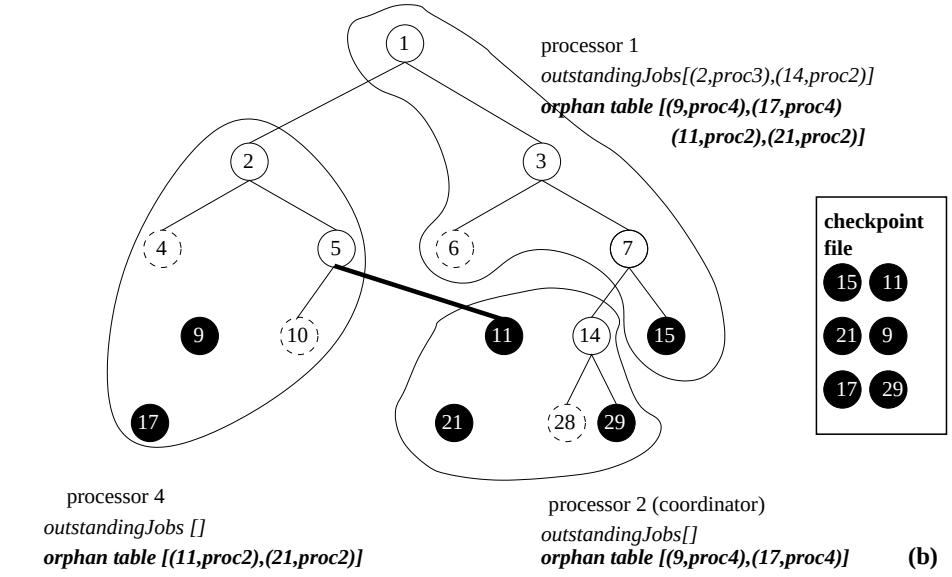
1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.13: Crash handling procedure and reading the checkpoint file





1 Job finished

1 Job in progress

1 Job spawned but not yet started

Figure 3.14: Reusing the checkpointed results

When a computation is started, the coordinator checks if the checkpoint file specified by the user already exists. If this is the case, the coordinator assumes that the computation has been restarted after an abort or a total crash. All results from the checkpoint file are read into the memory of the coordinator, and for each of those results a (jobID, processorID) tuple is sent to other processors. We use message combining to avoid sending each tuple separately.

Currently, all results read from the checkpoint file are stored in the memory of the coordinator. However, the amount of checkpointed data might be simply too large to fit in the memory of the coordinator. An alternative solution would be *distributing* the results among all the processors (currently) taking part in the computation.

3.5.3 The checkpoint file

The checkpoint file contains the partial results of the computation. The checkpoint file is accessed by the coordinator, but it need not necessarily be located on the coordinator's local filesystem. In fact, the user may specify an arbitrary location for the checkpoint file. The access to the checkpoint file is implemented using the Java GAT (Grid Application Toolkit) interface [3], a Java implementation of the GAT [23]. Java GAT provides a high-level API for grid applications. Among others, Java GAT provides an API for file operations that hides the complexity of the underlying infrastructure from the programmer. With GAT, the programmer only needs to specify the file name and location. The GAT takes care of selecting the appropriate protocol (e.g., FTP, SSH, HTTP, GridFTP etc.) and automatically optimizes the adjustable parameters based on available information on the current environment.

For data intensive applications, the checkpoint file might become huge. If the amount of space on stable storage is limited, it is necessary to prevent the checkpoint file from growing too much. Therefore, we implemented *checkpoint file compression*. During the application run, each checkpointed result eventually becomes redundant. This happens when the parent of the checkpointed job is also written to the checkpoint file. Therefore, the results of the children can be removed from the checkpoint file. However, we do not remove the children from the checkpoint file as soon as their parents are checkpointed, since this would cause much I/O overhead. Instead, compression is performed when the checkpoint file exceeds the size specified by the user. During the compression phase, a new checkpoint file is created and all the non-redundant results from the old file are written to the new file. Then, the old file is deleted. Note that the amount of free space on stable storage must be roughly twice as big as the maximal checkpoint file size specified by the user. In the rare case that the compression does not result in significant enough reduction of the checkpoint file size, checkpointing is stopped: no new results will be checkpointed. Checkpoint file compression is performed by the coordinator.

3.5.4 The coordinator

The coordinator is responsible for accessing the checkpoint file. The coordinator is elected from among the processors taking part in the computation. A simple approach

would be using the master as a checkpointing coordinator. However, to achieve the optimal performance, the I/O bandwidth and latency between the coordinator and the checkpoint file needs to be taken into account. Therefore, the processor with the best I/O performance is elected to be the coordinator. The election is performed in the following way.

1. The master is elected using the Registry
2. Each processor measures the time it takes to write a small file to the location where the checkpoint file will be created.
3. The results of those measurements are sent to the master.
4. The master waits until it receives such messages from at least 50% of the processors.
5. The master selects the processor with the shortest file write time and announces it as the new coordinator.

If the coordinator crashes, a new coordinator has to be elected. The new election is initiated by the master, which sends a *coordinator reelection* message to all processors. Then, the normal coordinator election procedure is performed. The processors postpone checkpointing until the election is completed.

If the coordinator has crashed while another process was sending checkpoint data to it, the data will be lost and never written to the checkpoint file. The loss of checkpoint data might affect the performance of the fault tolerance mechanism but not its correctness. Therefore, we do not take any action to avoid such situations.

The coordinator may also crash while writing to the checkpoint file and the checkpoint file may be corrupted. Therefore, each time a coordinator is initialized, it checks the checkpoint file (if it exist) for possible errors. If errors are found, it creates a new checkpoint file and transfers all non-damaged results from the old file to the new one. The old file is deleted.

To minimize the overhead of checkpointing, we use *concurrent checkpointing* [124]. The results are written to the checkpoint file by a separate thread in the coordinator process. This thread runs concurrently with the Satin computation.

3.6 Performance evaluation

In this section, we will evaluate the performance of our fault-tolerance algorithms. First, we evaluate the overhead of our algorithms during crash-free execution. Second, we evaluate the performance of our algorithms in the presence of crashes. We evaluate both the basic orphan-saving algorithm and the checkpointing extension with various checkpointing intervals. We will show that our algorithms add little overhead to Satin.

Next, we will show that our basic scheme outperforms the traditional approach, which does not save orphan jobs, and that using checkpointing further improves the performance.

	1 min	2 min	5 min
Raytracer	28 MB	20 MB	17 MB
TSP	217 KB	128 KB	55 KB

Table 3.1: Checkpoint file sizes

Further, we will show that our mechanism can be used for efficient migration of the computation. Finally, we will demonstrate that using the checkpointing extension, the computation can be stopped and restarted without losing work.

The experiments were carried out on the Distributed ASCI Supercomputer 2 (DAS-2). DAS-2 consists of five clusters located on five Dutch universities, in four Dutch cities: Amsterdam, Leiden, Delft and Utrecht. One of the clusters consists of 72 nodes, the others consist of 32 nodes, so there are 200 nodes in total. Each node contains two 1-GHz Pentium-IIIs and at least 1 GB RAM. All nodes run RedHat Linux. Within a single cluster, nodes are connected by Myrinet [48] and 100 Mb/s Ethernet. The clusters are interconnected by SurfNet, the Dutch university Internet backbone. The bandwidth between the sites ranges from 300 Mb/s to 1 Gb/s. The latencies are around 2ms.

All experiments described in this section were carried out on 32 nodes in 2 clusters (16 nodes in each cluster). For intra-cluster communication we used Ethernet.

In our experiments, we used the following applications:

- Raytracer which renders a picture (bitmap) using an abstract description of a scene. Raytracer has been parallelized by recursively subdividing the bitmap into smaller parts and rendering the parts in parallel. Raytracer is a relatively communication-intensive application.

- Traveling Salesman Problem (TSP) which searches for a shortest path connecting a set of cities. TSP is a well-known NP-complete problem which has many applications in science and engineering (e.g., manufacturing of circuit boards, analysis of the structure of crystals, clustering of data arrays, etc.). TSP was parallelized by evaluating different paths in parallel. The TSP implementation used in this evaluation is less efficient than industrial implementations. The reason is that the divide-and-conquer model does not allow data sharing between different subcomputations and therefore does not allow pruning of the search space[2]. However, our implementation is sufficient for the purpose of evaluating the performance of the fault-tolerance algorithms. TSP is a computation-intensive application and sends little data.

[2]In chapter 5, we will present a data-sharing extension of our programming model and describe a more efficient implementation of TSP

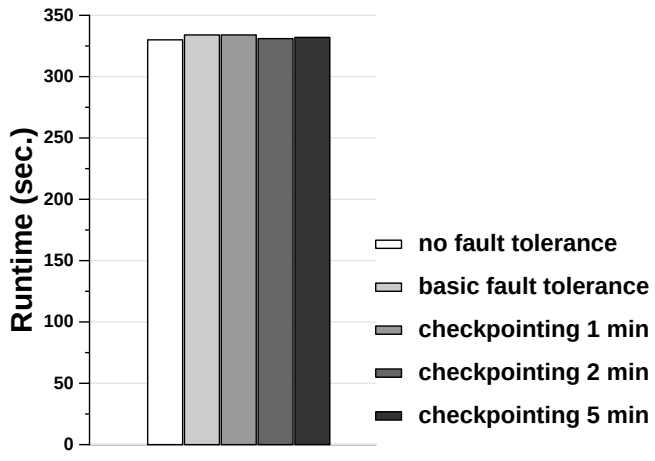


Figure 3.15: Raytracer, overhead during crash-free execution

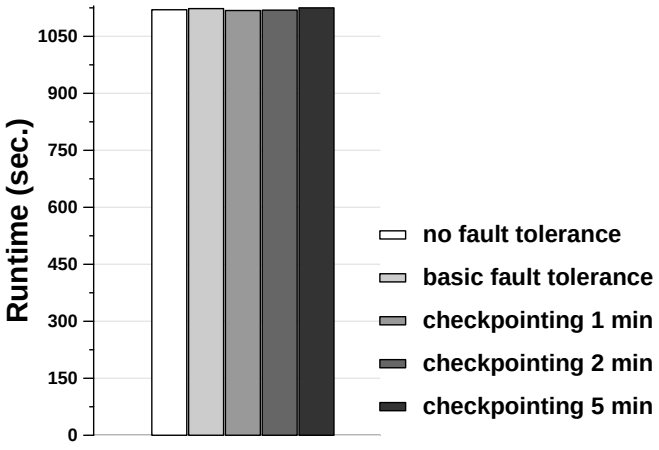


Figure 3.16: TSP, overhead during crash-free execution

3.6.1 Overhead during crash-free execution

In this section, we evaluate the impact of our algorithms on application performance when no processors are leaving or crashing. We run the applications in the following settings:

- The plain Satin system, that is, without any fault-tolerance mechanism enabled.
- The Satin system with the basic fault-tolerance mechanism (i.e., job recomputing and saving orphans) enabled.
- The Satin system with periodic checkpointing and with the following checkpointing intervals: 1, 2 and 5 minutes.

Figures 3.15 and 3.16 show runtimes of the two applications. The runtimes shown are averages over 2–4 runs. The standard deviations are around 2 seconds for Raytracer and 8 seconds for TSP.

The overhead of the basic fault-tolerance mechanism is negligible. Also, checkpointing has s small overhead and the overhead does not seem to be dependent on the checkpointing interval. This results from the fact that we are using concurrent checkpointing, which minimizes the impact of accessing the checkpoint file on the performance of the application. Table 3.1 lists the maximal sizes of the checkpoint files for different checkpoint intervals. The checkpoint files produced by the TSP application are small, since TSP does not process much data. Raytracer is more data intensive, and therefore produces larger checkpoint files.

3.6.2 Performance in the presence of crashes

In this section, we will evaluate the performance of our algorithms in the presence of crashes. First, we will compare the performance of our basic fault-tolerance algorithm (with orphan saving but no checkpointing) with the traditional ('naive') algorithm in which work lost in crashes is recomputed, but the orphans are not saved. Instead, orphans are discarded after computing them and recomputed later. Next, we will compare the performance of the checkpointing extension with the performance of the basic algorithm. We will look at different checkpointing intervals. Finally, we will evaluate the performance of our algorithm when the nodes are leaving *gracefully*, that is, after a prior notification.

In these experiments, we run the two applications on 32 nodes in 2 clusters. We remove one of the clusters in the middle of the computation, that is, after half of the time it would take on 2 clusters without processors leaving. The case when half of the processors leave is the most demanding, as the largest number of orphan jobs is created in this case. Typically, the number of orphans does not depend on the moment when processors leave, except for the initial and final phase in the computation.

To allow a fair comparison between various checkpointing intervals, we made sure that the crash happens always exactly in the middle of a checkpointing interval. We achieved it by adjusting the time the *first* checkpoint during the computation was taken. To compute the time of the first checkpoint, we used the following formula:

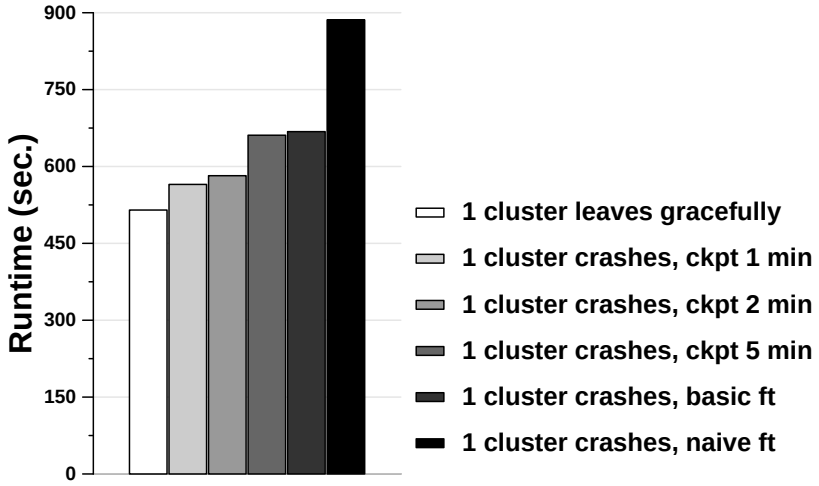


Figure 3.17: Raytracer, performance in the presence of crashes

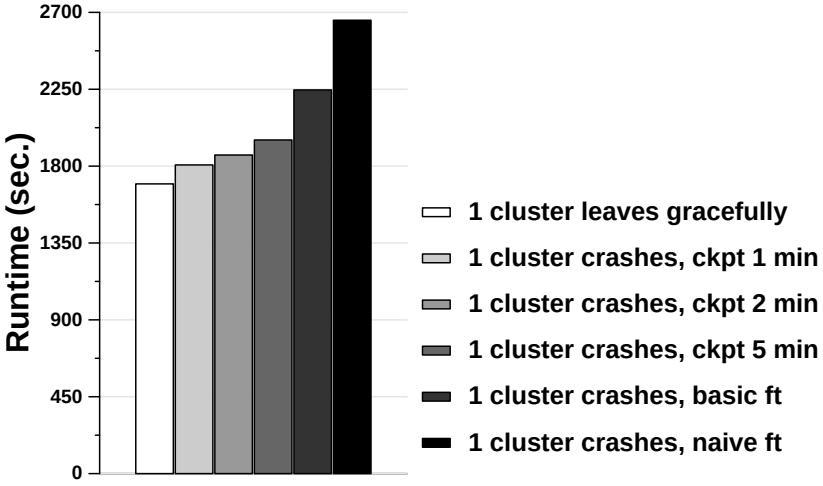


Figure 3.18: TSP, performance in the presence of crashes

	basic ft	ckpt 5 min	ckpt 2 min	ckpt 1 min	graceful
Raytracer					
jobs spawned	5 mln	5 mln	4.5 mln	4.5 mln	3.8 mln
jobs stolen	405	460	408	506	470
jobs in orphan tables	79	342	690	768	392
jobs reused	79	275	288	430	384
% jobs reused	100%	80%	42%	56%	98%
broadcast messages	11	22	22	29	25
TSP					
jobs spawned	400 000	360 000	330 000	330 000	290 000
jobs stolen	625	648	628	647	560
jobs in orphan tables	228	862	1503	2373	409
jobs reused	216	529	623	793	409
% jobs reused	95%	61%	41%	33%	100%
broadcast messages	12	16	20	25	14

Table 3.2: Orphan saving statistics

time of first checkpoint = (1/2 runtime on 32 cpus - 1/2 checkpoint interval) modulo checkpoint interval

The charts in figures 3.17 and 3.18 show the runtimes of both applications. The runtimes shown are averages taken over 4–6 runs. In 50% of the runs, the crashing (or leaving gracefully) cluster contained the master.

On average, our basic fault-tolerance algorithm outperforms the traditional, 'naive' approach by 15% to 25%. Checkpointing improves the performance of the system by further 10% to 15%. The performance improvement is largest with small checkpointing intervals. If nodes are leaving gracefully, the orphan saving algorithm provides up to 40% performance improvement over the 'naive' algorithm.

Table 3.2 lists average numbers of jobs stored in orphan tables and average number of jobs reused. While with the basic fault-tolerance algorithms almost all jobs are reused, when checkpointing is used, only 30% to 80% of jobs are reused. This is caused by the fact that many jobs in the checkpoint file are redundant, that is, their parent or other ancestor was checkpointed. In such cases, only the ancestor is used. Checkpoint compression can reduce the number of redundant jobs.

Table 3.2 also lists the number of broadcast messages sent in order to keep orphan tables up to date. Because message combining is used, this number is small and independent of the number of jobs in the orphan tables.

The variation in the runtimes for the traditional, 'naive' algorithm is large. This is caused by the fact that the performance of the traditional algorithm depends heavily on the number of orphan jobs created by the leaving processors, as all of those jobs have to be computed twice. Because work is distributed randomly, the variation in the number of created orphans is large which causes a large variation in runtimes for the traditional algorithm. Our algorithms are much less sensitive to the number of

	mean TSP	standard deviation TSP	mean Raytracer	standard deviation Raytracer
graceful	1695 s	1 s	514 s	27 s
ckpt 1 min	1806 s	119 s	565 s	25 s
ckpt 2 min	1865 s	175 s	582 s	37 s
ckpt 5 min	1953 s	139 s	661 s	36 s
ckpt 10 min	1971 s	108 s	687 s	73 s
basic ft	2246 s	258 s	668 s	25 s
naive ft	2654 s	649 s	886 s	205 s

Table 3.3: Crash performance statistics

orphans, as only small overhead is incurred by reusing orphans. Table 3.6.2 lists the standard deviations and means for all algorithms. These statistics were computed over 4–6 runs.

The difference between the 'naive' algorithm and our algorithm is biggest when the cluster containing master crashes. In that case, all the jobs become orphans and with the traditional approach, the computation must be started from the beginning. Our algorithm can reuse all the orphans and therefore the performance of the system stays the same regardless of whether the master crashes or not.

3.6.3 Performance of migration

In this section, we will evaluate the overhead of malleability based migration. In this experiment, we started an application on 32 nodes in 2 clusters. In the middle of the computation, we gracefully removed one of the clusters and replaced it with another cluster with the same number of processors (16). We compared the resulting runtime with a runtime without migration. These runtimes are shown in figures 3.19 and 3.20. The difference in the runtimes shows the overhead of migration. With our approach, the overhead is smaller than 5%. There are two sources of this overhead. First, the results from the leaving processors need to be sent over the network. Depending on the application, the amount of data to be sent can be significant. Second, part of the jobs need to be recomputed after migration, as only jobs that are finished at the moment the migration is requested are saved and transferred to other processor.

The overhead stays small, however, which shows that our mechanism can be used for efficient migration of the computation.

3.6.4 Performance of the abort/restore mechanism

In this section, we will evaluate the performance of the abort/restore mechanism. In this experiment, we ran an application on 32 nodes in 2 clusters. In the middle of the computation, we stopped the application by sending it an 'abort' signal. The application checkpointed its results and exited. Next, we have restarted the application on the same processor set and using the checkpoint file created in the aborted run.

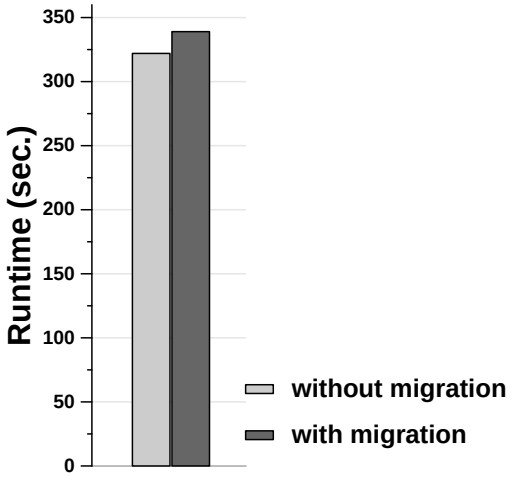


Figure 3.19: Raytracer, performance of migration

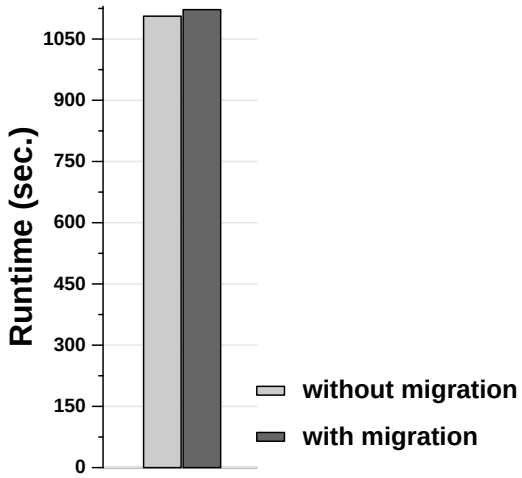


Figure 3.20: TSP, performance of migration

Figure 3.21: Raytracer, performance of abort/restore

Figure 3.22: TSP, performance of abort/restore

application	file size
Raytracer	12 MB
TSP	19 KB

Table 3.4: Checkpoint file size while aborting and restoring applications

We compared the resulting runtime with a runtime without abort/restore. Those runtimes are shown in figures 3.21 and 3.22. The overhead of aborting and restoring an application is 10% for a data intensive application (Raytracer) and only 1% for a computation intensive application (TSP). This overhead is caused by the need to write and read the checkpoint file. Practically no work is lost while aborting and restoring an application. The sizes of the checkpoint files are listed in table 3.4.

3.7 Comparison with related work

Several fault tolerance mechanisms designed specifically for divide-and-conquer applications have been proposed in the literature. An interesting approach was presented by Finkel and Manber in [83]. Their system, DIB, works in a way similar to Satin: it runs divide-and-conquer applications in parallel by executing subproblems on different processors. Load balancing is done by work stealing. The fault-tolerance mechanism is based on redoing of work. Processors in DIB redo work of other processors even if no crash has been detected. Redoing occurs while a processor waits for its steal request to be granted. Instead of staying idle, the processor starts redoing work that was stolen from it earlier but whose result it has not yet received. This approach is robust since crashes can be handled even without being detected. However, this strategy can lead to a large amount of redundant computation. The authors report the *ancestral-chain* problem in their paper: assume that process P1 gave some work to P2 which in turn gave some of it to P3, which failed before reporting the result back to P2. In that case both P1 and P2 will redo the work they gave away and the work given to P3 will be redone twice. Another problem, not discussed in the paper, are orphan jobs. Orphan jobs are not aborted after a crash was discovered, but executed until the end. When the result of an orphan is returned to its parent, it will be discarded, since the parent has crashed. The same job will be computed again while redoing the work given to the crashed processor. Therefore, like in the case of ancestral chains, part of the work will be done twice.

Another approach was proposed by Lin and Keller [126]. Similarly to the DIB approach, they base their fault tolerance mechanism on redoing the work. When a crash of a processor is detected, the jobs stolen by the crashed processor are redone by the owners of those jobs, i.e., the processors from whom the jobs were stolen. The authors try to handle the problem of orphan jobs. They achieve it by storing with each job not only the identifier of its parent processor (the processor from which the job was stolen), but also the identifier of its *grandparent* processor (the processor from which the parent processor stole the ancestor of our job). When the parent

processor crashes, the orphaned job is passed after completion to the grandparent processor which in turn passes it to the processor which is redoing the work lost in the crash. The result of an orphaned job can thus be reused. However, if both parent and grandparent processor crash, the orphaned job cannot be reused anymore. The concept can be extended by storing great-grandparent and higher level processor identifiers to be able to handle more crashes, but the number of crashes a specific implementation of this scheme can handle will always be limited by the number of pointers the implementation stores. Moreover, the amount of data that needs to be stored depends linearly on the number of crashes the implementation can handle. Another problem with this mechanism is that the result of an orphan job is passed to the grandparent processor only after the execution of this job is completed, which may occur a long time after the crash. By that time, some other processor may have already started or even completed redoing the same job. Our experiments show that such situations occur often. Therefore, although this mechanism tries to reuse orphan jobs, the amount of redundant work is still high.

Atlas [32] is another divide-and-conquer system. It was designed with heterogeneity and fault tolerance in mind and aims only at reasonable performance. Its fault tolerance mechanism is also based on redoing the work. The problem of orphan jobs is not addressed in Atlas. Atlas and its fault tolerance mechanism was based on CilkNOW [44] – an extension of Cilk [46], a C-based divide-and-conquer system. Cilk was designed to run on shared-memory machines while CilkNOW supports networks of workstations.

3.8 Conclusion

In this chapter, we presented a mechanism that enables fault tolerance, malleability and migration for divide-and-conquer applications. We proposed a novel approach to reusing partial results by restructuring the computation tree. Using this approach we minimized the amount of redundant computation, which is a problem of many other fault-tolerance mechanisms for divide-and-conquer systems. Our approach also allows to save almost all the work done by the leaving processors, when they leave gracefully. Divide-and-conquer applications using our mechanism can adapt to dynamically changing numbers of processors and migrate the computation between different machines without loss of work.

Further, we extended our basic fault-tolerance mechanism with a simple checkpointing facility. This extension allows the application to survive a total crash and improves the performance of crash recovery when a significant part of the processor has crashed. Finally, the checkpointing facility allows to abort an application and restart it without loss of work.

We implemented our algorithms in Satin and evaluated them on a wide-area DAS-2 system. In those experiments, we showed that the overhead of our algorithms during crash-free execution is very small. We also showed that when processors crash, our basic fault-tolerance algorithm outperforms the traditional approach (which does not reuse orphans) by 15 to 25%. Checkpointing can improve the performance by a

further 10%. Finally, when nodes leave gracefully the performance improvement of the orphan-saving approach over the traditional approach can reach 40%. We have also demonstrated the orphan-saving algorithm can be used for very efficient migration (with an overhead of smaller than 5%) and that the checkpointing facility can be used for aborting and restarting an application without loss of work.

Chapter 4

Self-adaptation

4.1 Introduction

One important problem in grid computing is *resource selection* – selecting a set of compute nodes such that the application achieves good performance. Even in traditional, homogeneous parallel environments, finding the optimal number of nodes is a hard problem and is often solved in a trial-and-error fashion. In a grid environment this problem is an order of magnitude harder because of the heterogeneity of resources: the compute nodes have various speeds and the quality of network connections between them varies from low-latency and high-bandwidth local-area networks (LANs) to high-latency and possibly low-bandwidth wide-area networks (WANs). Another important problem is that the performance and availability of grid resources varies over time: the network links or compute nodes may become overloaded, the compute nodes may become unavailable because of crashes or because they have been claimed by a higher priority application. Also, new, better resources may become available. To maintain a reasonable performance level, the application therefore needs to *adapt* to the changing conditions.

In this chapter, we will first discuss existing solutions to the resource selection and adaptation problems. Current approaches to the resource selection problem [172, 37] typically assume the existence of a *performance model* for an application – a mathematical formula that allows to predict the application runtime on a given set of resources. The performance model is used to evaluate a number of possible resources sets and choose the most appropriate one.

The adaptation problem can be reduced to the resource selection problem: the resource selection phase can be repeated during application execution, either at regular intervals, or when a performance problem is detected, or when new resources become available. A precondition here is that the application is *malleable* or *migratable*, that is, it can be moved to a different set of resources at runtime.

Constructing performance models for parallel applications is an inherently difficult task. Creating such a model requires not only application domain knowledge but also familiarity with complex parallel and distributed programming issues. In this chapter,

we will describe an approach to resource selection and adaptation which does not use performance models.

The rest of this chapter is structured as follows. In section 4.2, we will present background information on resource selection and adaptation. In section 4.3, we will describe our approach to resource selection and adaptation. In section 4.4, we will evaluate our approach, and in section 4.5, we will compare it with related work. We conclude in section 4.6.

4.2 Background

In this section, we will discuss some background on resource selection and application adaptation. We will describe the existing approaches to those problems.

4.2.1 Resource selection

The resource selection problem involves choosing a subset of the set of all available resources (compute nodes) on which the application will achieve a certain level of performance. Typically, a resource set that yields the *shortest* execution time is searched for. Alternatively, a resource set which allows the application to finish before a certain *deadline* is selected. Note that both of those approaches need a way of *predicting* the runtime of the application on a given set of resources.

In *economy based* grid computing [52] an extra search parameter is added: resource cost. The total cost of the selected resource set must fall within a user-defined *budget* and the application execution time should be minimized or the application must finish before a given deadline.

Finding the resource set that gives an optimal performance requires, in the most general case, an exhaustive search through all resource subsets. In the case of sequential applications, the complexity of the problem is $O(n)$ where n is the number of available resources, but in the case of parallel applications the problem is NP-complete (the number of possible subsets is 2^n). Since the number of available resources may be very large and the resource selector must deliver an answer within reasonable time, heuristics for *pruning* the search space are necessary. For example, in [65] resources are grouped into *clusters* (sets of processors such that network latencies within a set are lower than network latencies between the sets) and each possible set of such clusters is evaluated. For each set of clusters, machines are sorted according to a certain metric (three metrics are tried out for each cluster set: available memory, CPU speed and the combination of the two). Next, the first N machines from the sorted list are taken, for N ranging from 1 to the total number of machines in the cluster set, and the resulting resource set is evaluated. If it yields an execution time shorter that the current best set, it becomes the current best set. In [144], a greedy strategy is used: the collection of machines is extended in each step with a machine with the highest average bandwidth from all available machines. The procedure is repeated as long as the predicted execution time becomes shorter.

To select an appropriate set, a method of *ranking* the possible resource sets is

needed. One method is using a *performance model* which allows predicting the application running time on a given set of resources. Creating performance models is a challenging task. It requires knowledge not only of the application domain but also of the parallel computing issues. The literature describes such models only for relatively simple, regular applications, such as parameter sweeps [37], master-worker applications with homogeneous tasks [155] or regular iterative applications [128]. The performance model approach has been used in such projects as AppLeS [37] and GrADS [173].

Instead of using a detailed performance model of an application, some heuristic approach can be used. If only a single node needs to be selected (sequential applications), node ranking can be based on the node CPU speed (flops) [102]. Even though node speed does not always directly correspond to the application performance [142], node speed can be used as a heuristic replacing the use of a detailed performance model. This approach can be extended to parallel, *single-site* applications, i.e., parallel applications that can only run on a single cluster or supercomputer. Each site is ranked according to its number of nodes, node speed and average node load. The site with the biggest compute power is selected. This approach was used in the CactusCode project [21]. Heuristic approaches have not been used for applications running across multiple sites.

4.2.2 Adaptation

Grid environments are inherently dynamic. The availability and performance of grid resources is constantly changing. Even if an application is started on the *optimal* resource set, it may soon *become* suboptimal and the application performance may suffer. Therefore, to achieve optimal or even reasonable performance the application must constantly *adapt* to changing conditions. Application adaptation has two aspects: *when* to adapt, i.e., what circumstances should trigger the adaptation, and *how* to adapt, i.e. what actions should be taken to perform the adaptation.

Adaptation can be triggered by events such as:

- Application performance degradation.
- Availability of new resources that were not available at the moment the application was started.
- A change in application requirements.

To observe and measure the application performance degradation a concept of a *performance contract* was introduced. A performance contract specifies that given a set of resources with certain characteristics (e.g., bandwidth, processor speeds) an application will achieve a specified performance [150]. Application performance can be measured in a variety of ways. For example, a specified number of iterations per second needs to be achieved as in the CactusWorm experiment [21]. In [173], the real execution time of certain computation phases needs to be close to the execution time predicted by the performance model.

The application can react to changes in the environment in two ways:

- The application can change its behavior to use the current resources in a different way.

- The application can be rescheduled on a different set of resources (the new and the old resource sets can have a common subset).

Changing the application behavior can involve changing the mapping of the application tasks to the available resources. For example, an overloaded processor can get a lighter task. In [70], this strategy has been used to make a Successive Over-Relaxation (SOR) application adaptive: the allocation of matrix rows is periodically changed to adapt to a changing load of processors. Dynamic load balancing strategies, such as the CRS used by Satin, or heuristics used for scheduling parameter-sweep applications in the AppLeS project [57] make the application automatically adapt to changing processor loads.

An alternative way of changing the application behavior is changing the algorithm. For example, if its resources become overloaded, an application can start performing the calculation with lower accuracy, or if a network bandwidth diminishes, an application might start transferring pictures in a lower resolution.

The strategy of changing the application behavior cannot be applied to all types of applications. Especially, the algorithm change strategy is only suitable for a limited class of applications. Moreover, the algorithm change strategy is very difficult to apply automatically by the compiler or the runtime system. Usually, such a strategy has to be explicitly programmed by the application programmer.

Also, changing the application behaviour might not be sufficient to adapt to certain changes in the environment, for example extremely overloaded processors or networks or crashing processors. In that case, the application needs to be rescheduled on a new set of processors. Typically, when an application needs to be rescheduled, a new resource selection phase takes place. Possible resource sets are re-evaluated and the application is migrated to the current best set. This strategy is more generic: it can be applied to any type of application, provided that the application is migratable and/or malleable. However, a performance model for the application must be available.

4.3 Avoiding performance models

Most of the existing approaches to resource selection and adaptation assume that a performance model of an application is available. However, constructing performance models for parallel applications is an inherently difficult task. Such models exist for simple, regular applications. However, the divide-and-conquer applications we are dealing with exhibit much more complex behavior and we believe that creating performance models for such applications would be an extremely difficult task. In general, creating performance models requires expertise which a typical application programmer may not have. Creating such a model requires not only application domain knowledge but also familiarity with complex parallel and distributed programming issues.

In this chapter, we describe an alternative approach to application adaptation and resource selection. We start an application on *any* set of resources. Simple heuristics can be used to select this initial set of resources (e.g., select fast processors rather than slow ones) but no performance model is needed. During the application run, we collect statistics about the run and use them to estimate the resource requirements of the application. Our approach does not use any application-specific statistics, but look at metrics that can be applied to any parallel application: parallel efficiency, communication overhead, etc. Looking at those parameters we can conclude, for example, that there is not enough bandwidth in the system, or that there are more nodes than the application degree of parallelism would justify. Next, we *refine* the resource set the application is running on by adding and/or removing compute nodes. We repeat this procedure periodically, which allows us to adapt to changing conditions.

A major advantage of our approach is that it improves application performance in many different situations that are typical for grid computing. It handles all of the following cases:

- Automatically adapting the number of processors to the degree of parallelism in the application, even when this degree changes during the computation.
- Migrating (part of) a computation away from overloaded resources.
- Removing resources with poor communication links that slow down the computation.
- Adding new resources to replace resources that have crashed.

4.3.1 Application requirements

We studied the adaptation problem in the context of divide-and-conquer applications. However, we believe that our methodology can be used for other types of applications as well. In this section we summarize the assumptions about applications that are important to our approach. We also discuss how our approach can be extended to different types of applications.

The first assumption we make is that the application is *malleable*, i.e., it is able to handle processors joining and leaving the on-going computation. In chapter 3, we showed how divide-and-conquer applications can be made fault tolerant and malleable. Processors can be added or removed at any point in the computation with little overhead.

The second assumption is that the application can efficiently run on processors with different speeds. This can be achieved by using a dynamic load balancing strategy, such as work stealing used by divide-and-conquer applications [176]. Also, master-worker applications typically use dynamic load-balancing strategies (e.g., MW [95] described in section 2.2.2). We find it a reasonable assumption for a grid application, since applications for which the slowest processor becomes a bottleneck will not be able to efficiently utilize grid resources.

Finally, the application is insensitive to wide-area latencies. Our strategies could be extended to handle latency-sensitive applications. However, such applications cannot run efficiently on wide-area grids.

4.3.2 Resource model

We assume the following resource model. The applications are running on *multiple* sites at the same time, where each site is a cluster or supercomputer. We also assume that the processors of the sites are accessible using a grid scheduling system, such as Koala [136], Zorilla [72] or GRMS [23]. Processors belonging to one site are located on the same LAN. The communication between the processors on the same site is characterized by low latency and high bandwidth. Sites are connected by a WAN. Communication between sites suffers from high latencies. We assume that the links connecting the sites with the Internet backbone might become bottlenecks causing the inter-site communication to suffer from low bandwidths.

4.3.3 Weighted average efficiency

In traditional parallel computing, a standard metric describing the performance of a parallel application is *parallel efficiency*. Efficiency is defined as the average utilization of the processors, that is, the fraction of time the processors spend doing useful work rather than being idle or communicating with other processors [74].

$$efficiency = \frac{1}{n} * \sum_{i=0}^{n} (1 - overhead_i)$$

where n is the number of processors and $overhead_i$ is the fraction of time the i^{th} processor spends being idle or communicating. Efficiency allows calculating the application *speedup* which indicates the benefit of using multiple processors in comparison to using a single processor. The relationship between the efficiency and the speedup is expressed by the following formula:

$$efficiency = \frac{speedup}{n}$$

Typically, the efficiency drops as new processors are added to the computation. Therefore, achieving a high speedup (and thus a low execution time) and achieving a high system utilization are conflicting goals [74]. The optimal number of processors is the number for which the ratio of efficiency to execution time is maximized. Adding processors beyond this number yields little benefit. This number is typically hard to find, but in [74] it was theoretically proven that if the optimal number of processors is used, the efficiency is at least 50%. Therefore, adding processors when efficiency is smaller or equal to 50% will only decrease the system utilization without significant performance gains.

For heterogeneous environments, that is, environments with processors with different speeds, we extended the notion of efficiency and introduced *weighted average efficiency*.

$$wa\_efficiency = \frac{1}{n} * \sum_{i=0}^{n} speed_i * (1 - overhead_i)$$

In the above formula, the useful work done by a processor $(1 - overhead_i)$ is weighted average by multiplying it by the speed of this processor relative to the fastest processor. The fastest processor has $speed = 1$, for others holds: $0 < speed \leq 1$. Therefore, slower processors are modeled as fast ones that spend a large fraction of time being idle. Weighted average efficiency reflects the fact that adding slow processors yields less benefit than adding fast processors.

In the heterogeneous world, it is hardly beneficial to add processors if the efficiency is lower than 50% unless the added processor is faster than some of the currently used processors. Adding faster processors might be beneficial regardless of the efficiency.

4.3.4 Adaptation coordinator

In order to monitor the application performance and guide the adaptation, we added an extra process to the computation which we call *adaptation coordinator*. The adaptation coordinator periodically collects performance statistics from the application processors and computes the weighted average efficiency. If the weighted average efficiency falls above or below certain thresholds, the coordinator decides on adding or removing processors. A heuristic formula is used to decide which processors have to be removed. During this process the coordinator *learns* the application requirements by remembering the characteristics of the removed processors. Those requirements are then used to guide the adding of new processors.

4.3.5 Collecting performance statistics

Each processor measures the time it spends communicating or being idle. The computation is divided into *monitoring periods*. After each monitoring period the processors compute their overhead over this period as the percentage of the time they spent being idle or communicating in this period. Apart from the total overhead, each processor also computes the overhead of inter-cluster and intra-cluster communication.

In order to be able to calculate weighted average efficiency, we need to know the relative speeds of the processors. The speeds of the processors depend on the application and the problem size used. Since it is impractical to run the whole application on each processor separately, we use application-specific benchmarks. Currently we use the same application with a small problem size as a benchmark and we require the application programmer to specify this problem size. The disadvantage of this approach is that it requires extra effort from the programmer to find the right problem size and possibly produce input files corresponding to this problem size, which might be hard. An alternative solution would be generating benchmarks automatically by choosing a random subset of the task graph of the original application. For example in figure 4.1, two branches (darker nodes) of the execution tree are used as a benchmark.

Benchmarks have to be re-run periodically because the speed of a processor might change if it becomes overloaded by another application (for time-shared machines).

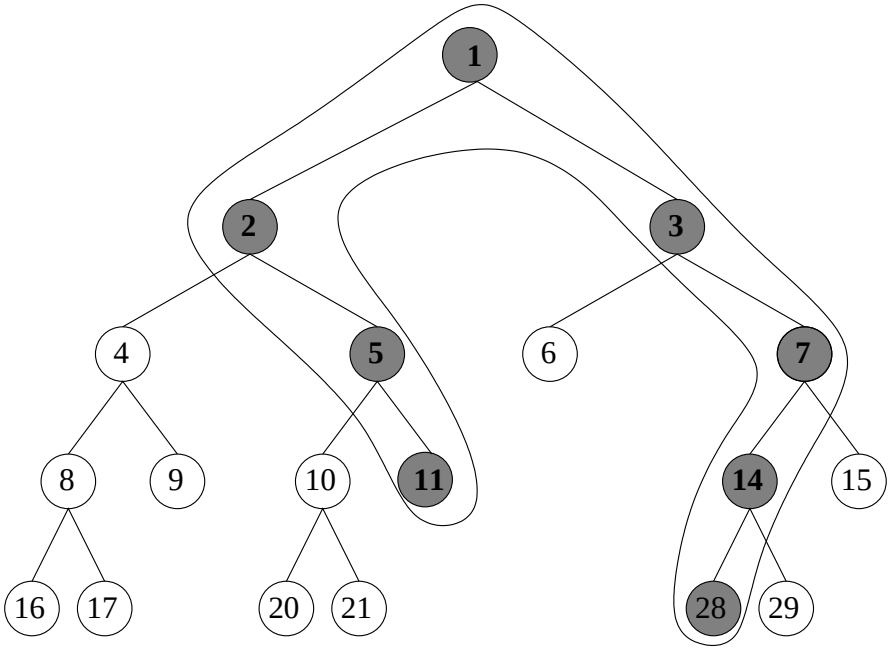


Figure 4.1: A subset of the execution tree used as a benchmark

Therefore, measuring the speed incurs an overhead. There is clearly a trade-off between the accuracy of speed measurements and the overhead it incurs. The longer the benchmark, the greater the accuracy of the measurement. The more often it is run, the faster changes in processor speed are detected. In our current implementation, the application programmer specifies the length of the benchmark (by specifying its problem size) and the maximal overhead it is allowed to cause. Processors run the benchmark at such frequency so as not to exceed the specified overhead. An improvement to this approach would be combining benchmarking with monitoring the load of the processor which would allow us to avoid running the benchmark if no change in processor load is detected. This optimization would further reduce the benchmarking overhead.

Note that the benchmarking overhead could be avoided completely for more regular applications, for example, for master-worker applications with tasks of equal or similar size. The processor speed could then be measured by counting the tasks processed by this processor within one monitoring period. Unfortunately, divide-and-conquer applications typically exhibit a very irregular structure. The sizes of tasks can vary by many orders of magnitude.

At the end of each monitoring period, the processors send the overhead statistics and processor speeds in this period to the coordinator. The adaptation coordinator stores the statistics received from the processors. Periodically, it computes the weighted average efficiency and other statistics, such as average inter-cluster overhead or overheads in each cluster. The clocks of the processors are not synchronized with each other or with the clock of the coordinator. Each processor decides separately when it is time to send data. Therefore, it happens occasionally that at the end of the monitoring period, the coordinator misses data from a few processors. In that case, the coordinator uses data from the previous monitoring period for those processors. This causes small inaccuracies in the calculations of the coordinator. In our experiments, we did not observe any influence of those inaccuracies on the performance of adaptation.

4.3.6 Adaptation strategy

The adaptation coordinator tries to keep the application weighted average efficiency between two thresholds: E_{min} and E_{max}. When the weighted average efficiency exceeds E_{max}, the adaptation coordinator requests new processors from the scheduler. The number of requested processors depends on the current efficiency: the higher the efficiency, the more processors are requested. The adaptation coordinator starts removing processors when the weighted average efficiency drops below E_{min}. The number of nodes that are removed depends on the weighted average efficiency. The lower the efficiency, the more nodes are removed. The thresholds we use are $E_{max} = 50\%$, because we know that adding processors when efficiency is lower does not make sense, and $E_{min} = 30\%$. Efficiency of 30% or lower might indicate performance problems such as low bandwidth or overloaded processors. In that case, removing bad processors will be beneficial for the application. Such low efficiency might also indicate that we simply have too many processors. In that case, removing some

processors may not be beneficial but it will not harm the application. The adaptation coordinator always tries to remove the 'worst' processors. The 'badness' of a processor is determined by the following formula:

$$proc\_badness_i = \alpha * \frac{1}{speed_i} + \beta * ic\_overhead_i + \gamma * inWorstCluster(i)$$

The processor is considered bad if it has low speed ($\frac{1}{speed}$ is big) and high inter-cluster overhead ($ic\_overhead$). High inter-cluster overhead indicates that the bandwidth to this processor's cluster is insufficient. Removing processors located in a single cluster is desirable since it decreases the amount of wide-area communication. Therefore, processors belonging to the 'worst' cluster are preferred. The function $inWorstCluster(i)$ returns 1 for processors belonging to the 'worst' cluster and 0 otherwise. The 'badness' of clusters is computed similarly to the 'badness' of processors:

$$cluster\_badness_i = \alpha * \frac{1}{speed_i} + \beta * ic\_overhead_i$$

The speed of a cluster is the sum of processor speeds normalized to the speed of the fastest cluster. The $ic\_overhead$ of a cluster is an average of processor inter-cluster overheads. The α, β and γ coefficients determine the relative importance of the terms. Those coefficients are established empirically. Currently we are using the following values: $\alpha = 1$, $\beta = 100$ and $\gamma = 10$, based on the observation that $ic\_overhead > 0.2$ indicates bandwidth problems and processors with $speed < 0.05$ do not contribute to the computation.

Additionally, when one of the clusters has an exceptionally high inter-cluster overhead (larger than 0.25), we conclude that the bandwidth on the link between this cluster and the Internet backbone is insufficient for the application. In that case, we simply remove the whole cluster instead of computing node badness and removing the worst nodes. After deciding which nodes are removed, the adaptation coordinator sends a message to those nodes, and the nodes leave the computation. Figure 4.2 shows a schematic view of the adaptation strategy. Dashed lines indicate a part that is not supported yet, as will be explained below.

This simple adaptation strategy allows us to improve application performance in several situations typical for the Grid:

- If an application is started on a smaller number of processors than its degree of parallelism allows, it will automatically expand to more processors (as soon as there are extra resources available). Conversely, if an application is started on more processors than it can efficiently use, a part of the processors will be released.

- If an application is running on an appropriate set of resources but after a while some of the resources (processors and/or network links) become overloaded and slow down the computation, the overloaded resources will be removed. After removing the overloaded resources, the weighted average efficiency will increase

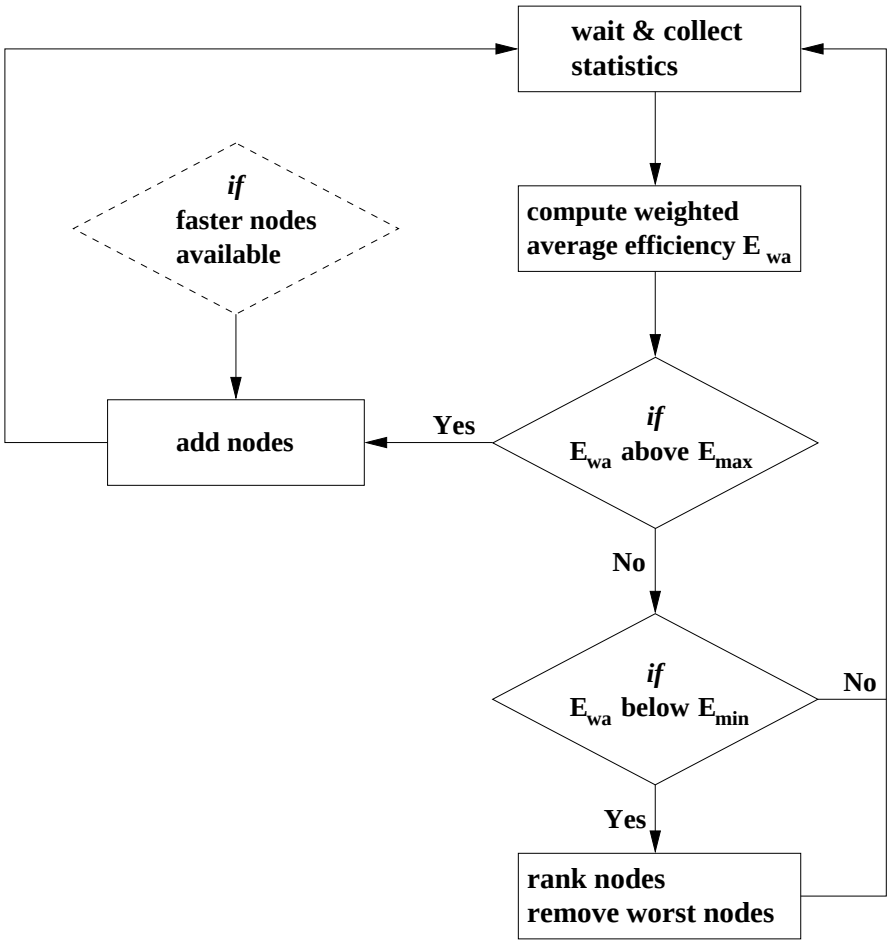


Figure 4.2: Adaptation strategy

to above the E_{max} threshold and the adaptation coordinator will try to add new resources. Therefore, the application will be *migrated* from overloaded resources.

- If some of the original resources chosen by the user are inappropriate for the application, for example the bandwidth to one of the clusters is too small, the inappropriate resources will be removed. If necessary, the adaptation component will try to add other resources.
- If during the computation a substantial part of the processors crash, the adaptation component will try to add new resources to replace the crashed processors.
- If the application degree of parallelism is changing during the computation, the number of nodes the application is running on will be automatically adjusted.

4.3.7 Further improvements of the adaptation strategy

Further improvements of our adaptation mechanism are possible, but require extra functionality from the grid scheduler and/or integration with monitoring services such as NWS [181]. For example, adding nodes to a computation can be improved. Currently, we add any nodes the scheduler gives us. However, it would be more efficient to ask for the *fastest* processors among the available ones. This could be done, for example, by passing a benchmark to the grid scheduler, so that it can measure processor speeds in an application specific way. Typically, it would be enough to measure the speed of one processor per site, since clusters and supercomputers are usually homogeneous. An alternative approach would be ranking the processors based on parameters such as clock speed and cache size. This approach is sometimes used for resource selection for sequential applications [102]. However, it is less accurate than using an application specific benchmark.

Also, during application execution, we can learn some application requirements and pass them to the scheduler. One example is the minimal bandwidth required by the application. The lower bound on minimal required bandwidth is tightened each time a cluster with high inter-cluster overhead is removed. The bandwidth between each pair of clusters is estimated during the computation by measuring data transfer times, and the bandwidth to the removed cluster is set as a minimum. Alternatively, information from a grid monitoring system can be used. Such bounds can be passed to the scheduler to avoid adding inappropriate resources. It is especially important when migrating from resources that cause performance problems: we have to be careful not to add the resources we have just removed. Currently we use *blacklisting* - we simply do not allow adding resources we removed before. This means, however, that we cannot use those resources even if the cause of the performance problem disappears, e.g. the bandwidth of a link might improve if the background traffic diminishes.

We are currently not able to perform *opportunistic migration* - migrating to better resources when they are discovered. If an application runs with efficiency between E_{min} and E_{max}, the adaptation component will not undertake any action, even if better resources become available. Enabling opportunistic migration requires, again,

the ability to specify to the scheduler what 'better' resources are (faster, with a certain minimal bandwidth) and receiving notifications when such resources become available. If that was possible, we could add those better resources even when we are running at good efficiency, and trigger removing (part of) the slower resources we are running on.

Existing grid schedulers such as GRAM from the Globus Toolkit [86] do not support such functionality. The developers of the KOALA metascheduler [136] have recently started a project whose goal is providing support for adaptive applications in KOALA. In the future, KOALA will provide the functionalities required by us to support opportunistic migration and to improve the initial resource selection.

4.3.8 Implementation

We instrumented the Satin runtime system to collect runtime statistics and send them to the adaptation coordinator. The coordinator is implemented as a separate process. For requesting new nodes, the Zorilla [72] system, described in section 2.2.1 is used. It allows straightforward allocation of processors in multiple clusters and/or supercomputers. Zorilla provides *locality-aware scheduling*. It tries to allocate processors that are located close to each other in terms of communication latency. In the future, Zorilla will also support bandwidth-aware scheduling, that is, a scheduling strategy that tries to maximize the total bandwidth in the system. Replacing Zorilla with another grid scheduler is straightforward. For example, Zorilla could be replaced with GAT [23] or KOALA [136].

4.4 Performance evaluation

In this section, we will evaluate our approach. We will demonstrate the performance of our mechanism in a few scenarios typical for grid environments. The first scenario is an 'ideal' situation: the application runs on a reasonable set of nodes (i.e., such that the efficiency is around 50%) and no problems such as overloaded network and processors, crashing processors, etc., occur. This scenario allows us to measure the overhead of the adaptation support. The remaining scenarios are typical for grid environments and allow us to demonstrate that with our adaptation support the application can avoid serious performance bottlenecks such as overloaded processors or network links.

For each scenario, we compare the performance of an application with adaptation support to a non-adaptive version. In the non-adaptive version, the coordinator does not collect statistics and or perform benchmarking (for measuring processor speeds). In the 'ideal' scenario, we additionally measure the performance of an application with collecting statistics and benchmarking turned on but without allowing it to change the number of nodes. This allows us to measure the overhead of benchmarking and collecting statistics. In all experiments we used a monitoring period of 3 minutes (180 seconds) for the adaptive versions of the applications.

All the experiments were carried out on multiple clusters of the DAS-2 wide-

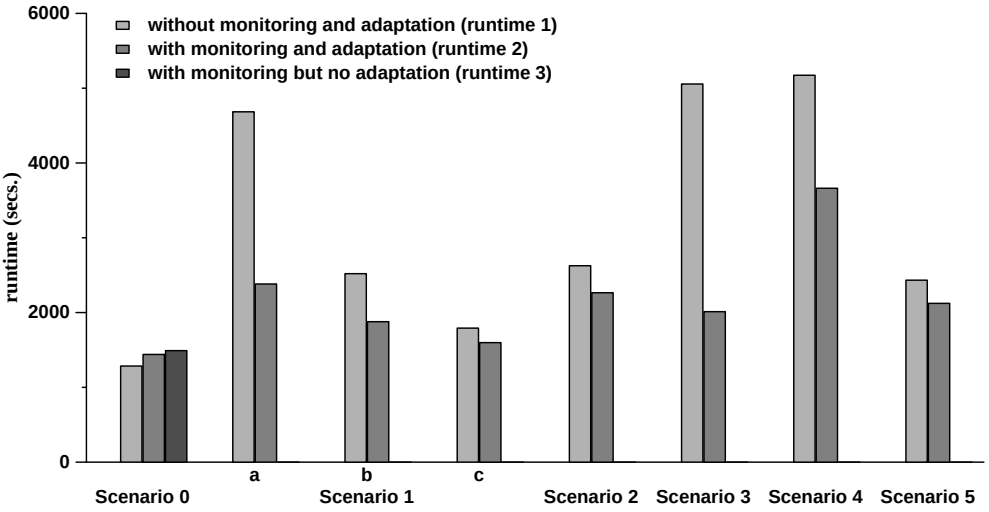


Figure 4.3: The runtimes of the Barnes-Hut application, scenarios 0-5

area system (DAS-2 was described in section 3.6). We used the Barnes-Hut N-body simulation. This application simulates the evolution of an N-body system under the influence of forces, for example gravitational or electrostatic forces. The simulation is carried out in discrete time steps (iterations). In each iteration the velocities of all bodies are computed and the positions of the bodies are adjusted[1].

We chose the Barnes-Hut simulation because it is an *iterative* application. Observing the variability in the iteration duration can give us more insight into the performance of the application under varying grid conditions and the effectiveness of adaptation.

4.4.1 Scenario 0: adaptivity overhead

In this scenario, the application is started on 36 nodes. The nodes are equally divided over 3 clusters (12 nodes in each cluster). On this number of nodes, the application runs with 50% efficiency, so we consider it a reasonable number of nodes. As mentioned above, in this scenario we measure three runtimes: the runtime of the application without adaptation support (runtime 1), the runtime with adaptation support (runtime 2) and the runtime with monitoring (i.e., collection of statistics and benchmarking) turned on but without allowing it to change the number of nodes (runtime 3). These runtimes are shown in figure 4.3, the first group of bars. The comparison between runtime 3 and 1 shows the overhead of adaptation support. In this experiment it is around 15%. Almost all overhead comes from benchmarking. The benchmark is run 1-2 times per monitoring period. This overhead can be made

[1] A more detailed description of the Barnes-Hut application can be found in section 5.6.3.

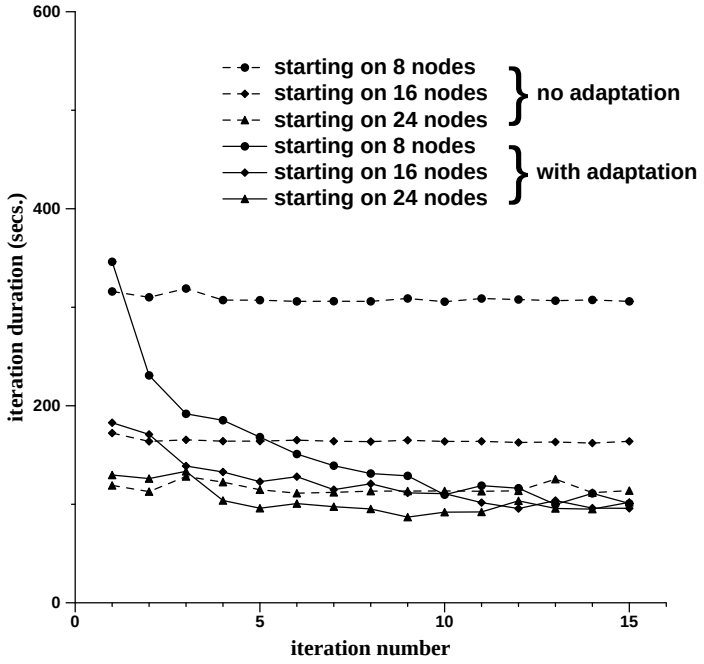


Figure 4.4: Barnes-Hut iteration durations with/without adaptation, too few CPUs (Scenario 1)

smaller by increasing the length of the monitoring period and decreasing the benchmarking frequency. The monitoring period we used (3 minutes) is relatively short, because the runtime of the application was also relatively short (approx. 30 minutes). Using longer running applications would not allow us to finish the experimentation in a reasonable time. However, real-world grid applications typically need hours, days or even weeks to complete. For such applications, a much longer monitoring period can be used and the adaptation overhead can be kept much lower. For example, with the Barnes-Hut application, if the monitoring period is extended to 10 minutes, the overhead drops to 6%. Note that combining benchmarking with monitoring processor load (as described in section 4.3.5) would reduce the benchmarking overhead in this scenario to almost zero: since the processor load is not changing, the benchmarks would only need to be run at the beginning of the computation.

Note that runtime 2 (with adaptation) is slightly shorter than runtime 3 (without adaptation). The reason is that during the run with adaptation turned on, a few nodes were added to computation when at some point the measured normalized efficiency dropped slightly below 50%.

4.4.2 Scenario 1: expanding to more nodes

In this scenario, the application is started on a number of nodes that is smaller than the application can efficiently use. This may happen because the user does not know the right number of nodes or because a bigger number of nodes was not available at the moment the application was started. We tried 3 initial numbers of nodes: 8 (Scenario 1a), 16 (Scenario 1b) and 24 (Scenario 1c). The nodes were located on 1 or 2 clusters. In each of the three sub-scenarios, the application gradually expanded to 36-40 nodes located in 4 clusters. This allowed to reduce the application runtimes by 50% (Scenario 1a), 35% (Scenario 1b) and 12% (Scenario 1c) with respect to the non-adaptive version. These runtimes are shown in figure 4.3. Since Barnes-Hut is an iterative application, we also measured the time of each iteration. These times are shown in figure 4.4. Adaptation reduces the iteration time by a factor of 3 (Scenario 1a), 1.7 (Scenario 1b) and 1.2 (Scenario 1c) which allows us to conclude that the gains in the total runtime would be even bigger if the application were run for more than 15 iterations.

4.4.3 Scenario 2: overloaded processors

In this scenario, we started the application on 36 nodes in 3 clusters. After 200 seconds, we introduced a heavy, artificial load on the processors in one of the clusters. Such a situation might happen when an application with a higher priority is started on some of our resources. Figure 4.5 shows the iteration durations of both the adaptive and non-adaptive versions. After introducing the load, the iteration duration increased by a factor of 2 to 3. This happened in iteration 2 for the adaptive version and iteration 3 for the non-adaptive version (since the iterations in the non-adaptive version are slightly shorter). Also, the iteration times became very variable. The adaptive version observed a very low weighted average efficiency (20%) and reacted by removing the overloaded nodes (iteration 3). After removing these nodes, the weighted average efficiency rose to around 65% which triggered adding new nodes (iteration 5) and the application expanded back to 38 nodes. So, the overloaded nodes were replaced by better nodes, which brought the iteration duration back to the initial value. This reduced the total runtime by 14%. The runtimes are shown in figure 4.3.

4.4.4 Scenario 3: overloaded network link

In this scenario, we ran the application on 36 nodes in 3 clusters. We simulated that the uplink to one of the clusters was overloaded and the bandwidth on this uplink was approximately 100 KB/s.

To simulate low bandwidth we use the traffic-shaping techniques described in [63][2]. To achieve the specified sending rate, the sender *sleeps* an appropriate time between sending packets. The sleeping time is calculated as a difference between the time the transmission should have taken if the link had the specified bandwidth and the time the transmission really took. This is done both on the sending and on the receiving

[2]We used a traffic shaper implemented by Mathijs den Burger [68]

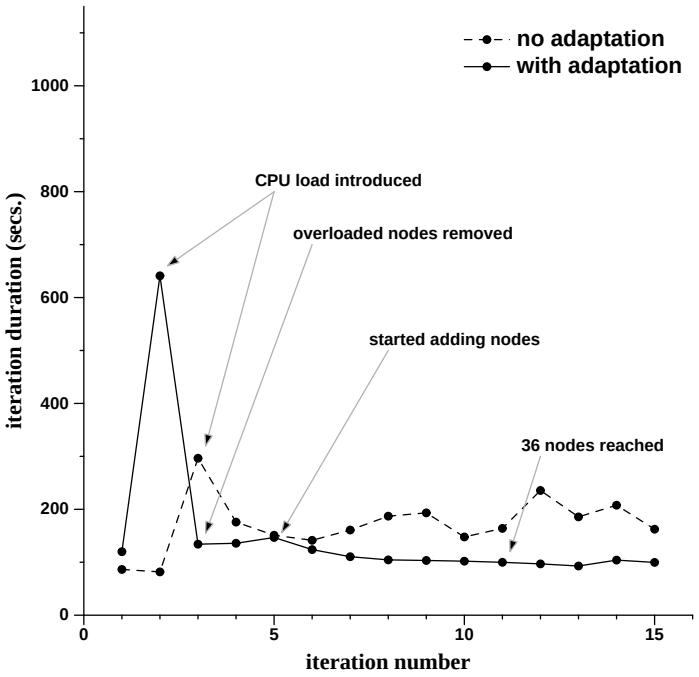


Figure 4.5: Barnes-Hut iteration durations with/without adaptation, overloaded CPUs (Scenario 2)

side. Care needs to be taken to deal with the coarse granularity of the sleep function. More details can be found in [63].

The iteration durations in this experiment are shown in figure 4.6. The iteration durations of the non-adaptive version exhibit enormous variation: from 170 to 890 seconds. The adaptive version observed a weighted average efficiency of 25% and a high WAN communication overhead in one of the clusters (40%). Therefore it removed the badly connected cluster after the first monitoring period. As a result, the weighted average efficiency rose to around 65% and new nodes were gradually added until their number reached 38. This brought the iteration times down to around 100 seconds. The total runtime was reduced by 60% (figure 4.3).

4.4.5 Scenario 4: overloaded processors and an overloaded network link

In this scenario, we ran the application on 36 nodes in 3 clusters. Again, we simulated an overloaded uplink to one of the clusters. Additionally, we simulated processors with heterogeneous speeds by inserting a relatively light artificial load on the processors in one of the remaining clusters. The iteration durations are shown in figure 4.7. Again,

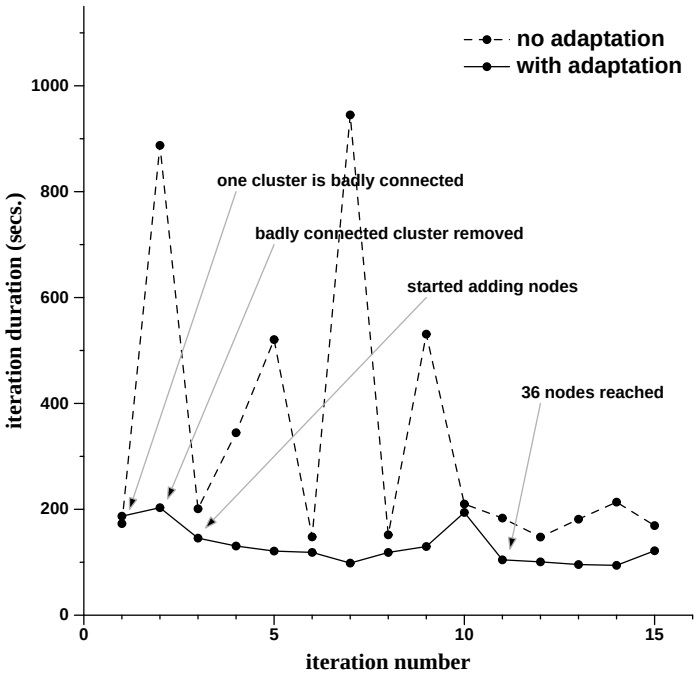


Figure 4.6: Barnes-Hut iteration durations with/without adaptation, overloaded network link (Scenario 3)

the non-adaptive version exhibits a great variation in iteration durations: from 200 to 1150 seconds. The adaptive version removes the badly connected cluster after the first monitoring period, which brings the iteration duration down to 210 seconds on average. After removing one of the clusters, since some of the processors are slower (approximately 5 times), the weighted average efficiency rises only to around 35-40% and oscilates around those values. At some point it drops slightly below 30% which triggers removing 2 of the slower nodes. This example illustrates what the advantages of *opportunistic migration* would be. There were faster nodes available in the system. If those nodes were added to the application (which could trigger removing more of the slower nodes) the iteration duration could be reduced even further. Still, the adaptation reduced the total runtime by 30% (figure 4.3).

4.4.6 Scenario 5: crashing nodes

In the last scenario, we also ran the application on 36 nodes in 3 clusters. After 500 seconds, 2 out of 3 clusters crash. The iteration durations are shown in figure 4.8. After the crash, the iteration duration rose from a 100 to 200 second. The weighted average efficiency rose to around 70%, which triggered adding new nodes in the adaptive version. The number of nodes gradually went back to 36, which brought the

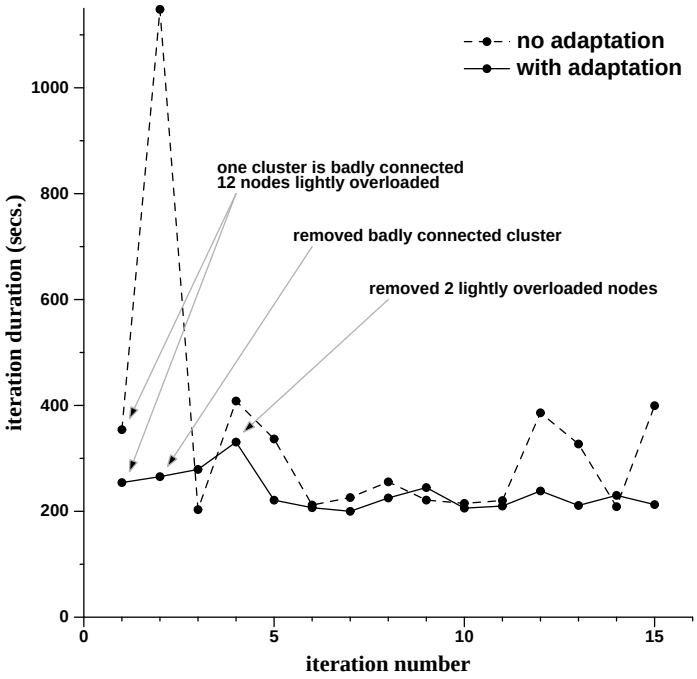


Figure 4.7: Barnes-Hut iteration durations with/without adaptation, overloaded CPUs and an overloaded network link (Scenario 4)

iteration duration back to around 100 seconds. The total runtime was reduced by 13% (figure 4.3).

4.5 Comparison with related work

A number of Grid projects address the question of resource selection and adaptation. In most of these projects, resource selection and adaptation depend on performance models that allow predicting application runtime on a given resource set. The Grid Application Development System (GrADS) [172] uses performance models to select the set of resources with the minimal predicted runtime. During the computation, the application performance is monitored using the Autopilot infrastructure [151]. If the ratio between the predicted and the actual application performance exceeds a certain threshold, migration is requested. Upon a migration request, the resource selection phase is repeated - possible resource sets are re-evaluated and if a better set of resources is found, migration is considered. A distinguishing feature of the GrADS environment is that it takes into account the remaining execution time of the application when considering migration. Migration is performed only when the predicted remaining execution time on the new set of resources plus the worst case migration

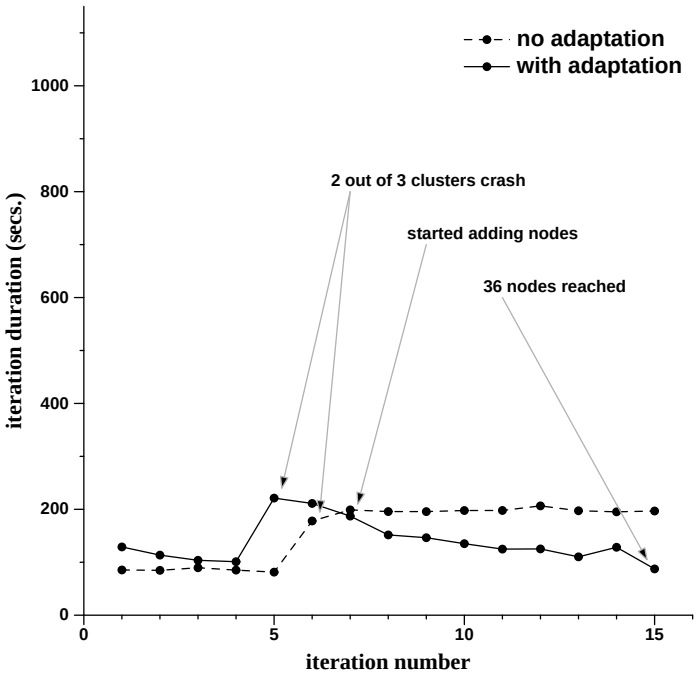


Figure 4.8: Barnes-Hut iteration durations with/without adaptation, crashing CPUs (Scenario 5)

time is smaller than the predicted remaining execution time on the current set of resources. This approach allows to avoid costly migrations when the application is close to completion. GrADS also supports opportunistic migration. If some other application has recently completed, the GrADS rescheduler determines whether performance benefits can be obtained for a currently executing application by migrating it to use the resources freed by the completed application.

The main difference between the GrADS environment and our approach is the use of performance models. The main advantage is that once the performance model is known, the system is able to take more accurate migration decisions than with our approach. However, even if the performance model is known, the problem of finding an *optimal* resource set (i.e. the resource set with the minimal execution time) is NP-complete. Currently, GrADS examine only a subset of all possible resource sets and therefore there is no guarantee that the resulting resource set will be optimal. As the number of available grid resources increases, the accuracy of this approach diminishes, as the subset of possible resource sets that can be examined in a reasonable time becomes smaller.

Unlike GrADS we are not able to predict the remaining execution time and take it into account when deciding on adaptation. For divide-and-conquer this is of lit-

tle importance, however, since adding and removing resources to divide-and-conquer computations has small overhead. GrADS supports opportunistic migration while our implementation currently does not. However, we plan to add support for opportunistic migration in the future. Finally, GrADS is suitable for iterative MPI applications while we are targeting at divide-and-conquer applications.

Cactus is a Grid-enabled computational framework for the construction of parallel solvers for partial differential equations. Cactus is suitable only for single-site (supercomputer or cluster) applications. No performance model is used. The available sites are ranked and the site with the highest rank is selected for execution. The rank of a site is its number of processors multiplied by the processor speed. The application can be migrated if a higher-ranked site is discovered or a performance degradation is observed. The application performance is expressed as the number of application iterations per second. The main difference between the Cactus methodology and our approach is that Cactus is suitable for single-site applications. For such applications, the complexity of the resource selection and adaptation problems is many orders of magnitude smaller than for multi-site applications: the set of possible resource sets is much smaller, the bandwidth between the sites does not have to be taken into account etc. Moreover, resource selection based on clock speed is not always accurate. Finally, performance degradation detection is suitable only for iterative applications and cannot be used for irregular computations such as search and optimization problems. We use performance degradation detection based on weighted average efficiency which can be applied to any parallel application.

The GridWay framework [102] has many similarities with the Cactus approach. It is targeted at sequential applications. In the resource selection phase, not only the speed of a candidate host but also its proximity to the application files, checkpoint files and the current host (in case of migration) is taken into account. Migration is performed when a better host is discovered or when performance degradation is detected. The application performance can be measured, for example, by counting the number of application iterations per second. The main differences with our approach are that we target multi-cluster, parallel applications while GridWay supports only sequential ones. Also, GridWay's performance degradation method is suitable only for iterative applications.

The resource selection problem was also studied by the AppLeS project [37]. In the context of this project, a number of applications were studied and performance models for those applications were created. Based on such a model a scheduling agent is built that uses the performance model to select the best resource set and the best application schedule on this set. AppLeS scheduling agents are written on case-by-case basis and cannot be reused for another application. Two reusable *templates* were also developed for specific classes of applications, namely master-worker (AMWAT template) and parameter sweep (APST template) applications. Migration is not supported by the AppLeS software.

In [100], the problem of scheduling iterative master-worker applications is studied. The authors assume homogeneous processors (i.e., with the same speed) and do not take communication costs into account. Therefore, the problem is reduced to finding the right number of workers. The approach here is similar to ours in that

no performance model is used. Instead, the system tries to deduce the application requirements at runtime and adjusts the number of workers to approach the ideal number. The adjustment is done on a per-iteration basis: the observations from the previous iteration are used to adjust the number of workers for the following iteration. Our approach supports a much wider variety of scenarios, i.e., heterogeneous node and network speeds. Also, our approach does not assume that the application is iterative.

Aldinucci et al. [19] present an abstract model of activities that need to be performed in order to handle adaptivity in distributed applications. They apply this model to the ASSIST framework for creating high-level, component-based applications. An ASSIST application consists of multiple modules which can themselves be parallel programs. It is possible to specify a Quality of Service contract for each module or for the whole application (similar to performance contracts in GrADS). If such a QoS contract is violated, adaptation is performed. The adaptation strategy for a component is based on the performance model of this component. ASSIST can automatically provide performance models for components that have a master-worker or a data-parallel structure. For other types of components, a performance model must be provided by the user.

4.6 Conclusion

In this chapter, we investigated the problem of resource selection and adaptation in grid environments. Existing approaches to those problems typically assume the existence of a performance model that allows predicting application runtimes on various sets of resources. However, creating performance models is inherently difficult and requires knowledge about the application.

We proposed an approach that does not require in-depth knowledge about the application. We start the application on an arbitrary set of resources and monitor its performance. The performance monitoring allows us to learn certain application requirements such as the number of processors needed by the application or the application's bandwidth requirements. We use this knowledge to gradually refine the resource set by removing inadequate nodes or adding new nodes if necessary. This approach does not result in the optimal resource set, but in a *reasonable* resource set, i.e. a set free from various performance bottlenecks such as slow network connections or overloaded processors. Our approach also allows the application to adapt to the changing grid conditions.

We implemented this approach in the Satin framework. We added an extra process called the *adaptation coordinator*, which collects the runtime statistics (i.e. the idle time, the local and wide-area communication time) and decides on adding or removing nodes. The decisions are based on the weighted average efficiency – an extension of the concept of parallel efficiency defined for traditional, homogeneous parallel machines. If the weighted average efficiency drops below a certain level, the adaptation coordinator starts removing 'worst' nodes. The 'badness' of the nodes is defined by a heuristic formula. If the weighted average efficiency raises above a certain level, new nodes are

added.

This simple strategy allows us to handle multiple scenarios typical for grid environments: expand to a bigger number of nodes or shrink to a smaller number of nodes if the application was started on an inappropriate number of processors, remove inadequate nodes and replace them with better ones, replace crashed processors, avoid slow networks, etc. The application adapts *fully automatically* to changing conditions. We tested our approach on the DAS-2 distributed supercomputer and demonstrate that our approach can yield significant performance improvements (up to 60% in our experiments).

Future work will involve extending our adaptation strategy to support opportunistic migration. This, however, requires grid schedulers with more sophisticated functionality than the functionality of the existing schedulers. Further research is also needed to decrease the benchmarking overhead. For example, the information about CPU load could be used to decrease the benchmarking frequency. Another line of research that may be investigated is using *feedback control* to refine the adaptation strategy during the application run. For example, the node 'badness' formula could be refined at runtime based on the effectiveness of the previous adaptation decisions. Finally, the centralized implementation of the adaptation coordinator might become a bottleneck for applications which are running on very large numbers of nodes (hundreds or thousands). This problem can be solved by implementing a *hierarchy* of coordinators: one sub-coordinator per cluster which collects and processes statistics from its cluster and one main coordinator which collects the information from the sub-coordinators.

Chapter 5

Data sharing in dynamic environments

5.1 Introduction

An important disadvantage of the divide-and-conquer paradigm is its limited applicability due to the lack of global state. The only way of sharing data between divide-and-conquer tasks is by explicit parameter passing. This model is insufficient for many applications [107]. One class of such applications consists of programs that pass large data structures as parameters. With pure divide-and-conquer, those large parameters need to be copied each time a task is executed remotely, while copying the parameters once and reusing them later would be more efficient. Another class of applications consists of programs which need to share data between independent tasks. In pure divide-and-conquer, this form of data sharing is not possible. Branch-and-bound applications belong to this class. Sharing the best known solution between all the processors taking part in the computation allows pruning large parts of the search trees. Another example is game-tree search where a transposition table is shared to avoid evaluating the same position twice.

In this chapter, we will extend the divide-and-conquer model with a shared data abstraction – shared objects. We will call the extended model *divide-and-share*. Implementing a shared data abstraction in a distributed system is a challenging problem. Providing a strong form of consistency (e.g., sequential consistency [119]) while maintaining high performance is infeasible even on tightly connected systems like clusters of workstations. In grid environments this problem is even harder. One problem is the high wide-area latencies. Another problem is that grids are inherently dynamic. The set of processors on which the application is running constantly changes. Most consistency protocols have been designed with a fixed set of processors in mind. Dynamic processor sets make consistency more complicated and expensive and therefore impractical for grid environments.

Many relaxed consistency models have been proposed (e.g., causal consistency [104],

DAG-consistency [45]), but none of them are suitable for grid-enabled divide-and-conquer grid applications, as they are either too expensive to implement in grid environments, or do not fit the needs of our applications.

Therefore, we designed a new, relaxed consistency model, which we call *guard consistency*. A programmer can define the consistency requirements of an application by means of *guard functions*. A guard function is associated with a divide-and-conquer task and defines whether the shared objects accessed by this task are in a consistent state. The runtime system allows the replicas to become inconsistent as long as the guards are satisfied. If a guard is not satisfied, the runtime system brings the local replicas into a correct state.

The rest of this chapter is structured as follows. In section 5.2, we present background information on data sharing. In section 5.3, we describe the shared objects model. In section 5.4, we describe the shared objects API and illustrate it with a number of code examples. In section 5.5, we describe the implementation of the shared objects model. In section 5.6, we discuss our experiences with programming applications with the new model. In section 5.7, we evaluate the performance of our model, and in section 5.8, we compare it with related work. We conclude in section 5.9.

5.2 Background

Shared data is an attractive model for expressing communication and synchronization in distributed applications. It is at a higher level of abstraction than explicit message passing and therefore significantly simplifies programming and debugging distributed applications. In this section, we will present background information on data sharing in distributed systems. We will discuss different programming models using the shared data abstraction: Shared Virtual Memory, shared object models and distributed tuple space models. Next, we will discuss the algorithms used to implement shared data abstractions. Finally, we will discuss the problem of shared data consistency and review a number of consistency models.

5.2.1 Shared data paradigms

Data sharing paradigms can be roughly divided into two categories: *unstructured* and *structured* paradigms [122]. Unstructured paradigms present the programmer with a flat address space similar to how the actual physical memory is seen by applications. With structured paradigms, the shared data is organized into user-defined abstract data structures. In this section we will describe both the unstructured (the Shared Virtual Memory) and structured approaches (shared objects and tuple spaces). The classification of shared data paradigms is shown in figure 5.1.

Shared Virtual Memory

Shared Virtual Memory (SVM) [123] simulates a real physical shared memory: the processes have an illusion of seeing a single shared address space. Processes can access the shared memory using simple *read*, *write* and *lock* operations.

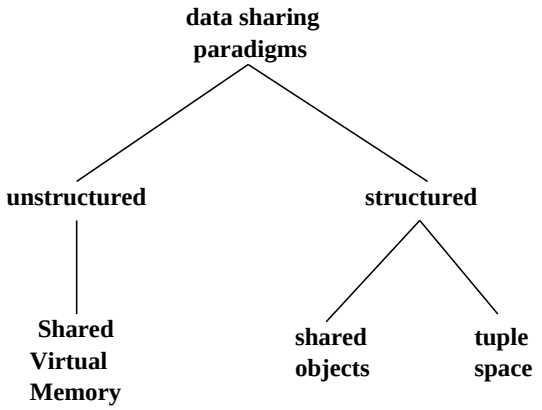


Figure 5.1: Data sharing paradigms

The address space of a SVM is partitioned into pages (blocks, segments). When a processor tries to access a page which is not present in its physical memory, the operating system or runtime system fetches the page from a remote processor and stores a copy of the page in the local memory.

The *granularity* of data sharing, that is the size of the page, varies in different systems. In some systems the unit of sharing is a multiple of the hardware page size (Ivy [123]), in others the unit of sharing is much smaller, for example 32 bytes (Memnet [67]). The choice of the granularity can have a large impact on the performance of the system. If the granularity is too small, many page transfers might occur within a short period. However, if the granularity is large, the probability of *false sharing* increases. False sharing occurs when two variables used by two different processes are allocated on one page. In this case, the page will be constantly moved between the two processes even though the variables are not shared. This problem results from the fact that the structure of the shared memory does not reflect the structure of the application. Therefore, Shared Virtual Memory is called an *unstructured* Distributed Shared Memory (DSM) [122].

Ivy [123] was the first implementation of Shared Virtual Memory. Later, many other systems were implemented, for example Memnet [67], Mirage [84], Plus [42], Shiva [125], TreadMarks [113], Mether [135], Mermera [98], Munin [36] etc. (see [140] for a detailed overview of a part of those systems). Shared Virtual Memory systems were targeted at tightly-coupled systems, such as multicomputers or small networks of workstations (typically 8 nodes, in some cases up to 64).

To avoid the mismatch between the structure of the application and of the shared data, *structured* shared data models were introduced [122]. In structured data models, the shared data appears to the application as a set of user-defined data structures. In the following sections, we will describe two structured approaches to data sharing: shared objects and tuple spaces.

Shared objects

Structured approaches to data sharing allow tailoring the granularity of data sharing to the application needs. Many such systems use the concept of a shared data-object or a shared object. Shared objects encapsulate shared data. They are user-defined abstract data structures that can be accessed by user-defined operations.

Encapsulating shared data into objects has many advantages. Sharing granularity depends on the application structure, as the unit of data sharing is a shared object instead of a page. This excludes the possibility of false sharing. Shared data is accessed using high-level, composite operations rather than low-level read/write operations, which reduces the communication overhead. Finally, shared objects allow synchronizing accesses to the shared data. The runtime system can take care that the high-level operations are executed indivisibly. This simplifies the programming task as the programmer does not need to use semaphores or locks [31].

Many Distributed Shared Memory systems based on the shared object model have been implemented. Some examples include: Orca [31], CRL [116], DiSOM [58], Amber [59], SAM [153], Agora [41], Clouds [148], GARF [147], Emerald [109], RepMI [130] and many others.

Tuple space

Another structured approach to data sharing is the concept of a *tuple space*. This concept was first introduced in the parallel language Linda [18]. Tuple space is a *distributed datastructure*, that is a datastructure that can be modified by multiple processes. A tuple space consists of *tuples* – ordered sequences of values. There are three operations that can be performed on tuples: *out*, *in* and *read*. *Out* adds a tuple to the tuple space. *In* reads a tuple and removes it from the tuple space. *Read* reads a tuple without removing it from the tuple space. Tuple space is an *associative memory*, meaning that the tuples do not have addresses but they are denoted by the values they contain.

Tuples residing in the tuple space are immutable. The only way of modifying a tuple is by taking it out of the tuple space, modifying it in the local memory of a processor and putting it back into the tuple space. This provides a natural way of serializing operations on the tuple space: if two or more processes want to modify the same tuple, only one process will succeed in taking the tuple out of the tuple space. The remaining processes will block until the first process finishes its modifications and puts the tuple back in the tuple space. However, this model might be inefficient if tuples contain large amounts of data, as in that case the whole tuple needs to be sent back and forth.

Many implementations of tuple spaces have been developed. Besides the implementation in the Linda programming language, implementations for Java (JavaSpaces [87] and TSpaces [121]), Smalltalk [132] and SML [156] exist.

		single−writer (primary−copy)		**multiple−writer**
		static owner	**migrating ownership**	
single−reader		Munin	Munin	
multiple−reader (replication)	**data shipping**	Ivy, Linda, CRL	Ivy	TreadMarks
	function shipping			Orca, RepMI

Figure 5.2: Algorithms implementing data sharing

5.2.2 Algorithms implementing data sharing

In this section, we will discuss a number of algorithms implementing shared data abstractions. Such algorithms can be divided in three categories: single-reader/single-writer, multiple-readers/single-writer and multiple-readers/multiple-writers [1] [146] (a similar classification can be found in [161]). In the single-reader/single-writer protocols, only one copy of each data item exists in the system. In the multiple-reader/* protocols, the data items are *replicated*, i.e. they exist in multiple copies. An overview of the algorithms discussed in this section and example systems using those algorithms are shown in figure 5.2.

In the single-reader/single-writer type algorithms, only one processor at a time has a copy of the data. We call such a process the *owner* of the data. It can be a static manager process (i.e., the owner of the data does not change during the computation) or the data might be migrated between the processes.

With the static manager approach, all operations on the shared data are forwarded to the manager which applies the operations on the data and sends back the results. This approach has two major drawbacks. First, each operation on the data performed by a process other than the manager requires communication over the network. Therefore, this approach is suitable only for tightly-coupled systems with low network latencies or for applications which access shared data infrequently. Another

[1] The */single-writer algorithms are also known as primary-copy algorithms

drawback of the central manager approach is the fact the manager will become a bottleneck if access to data is performed frequently. This problem can be alleviated by partitioning the data and assigning a different manager to each partition.

With data-migration, data is migrated to the processor that needs to access this data. The advantage of this approach over the previous one is that when the memory accesses exhibit good locality, only the first in a series of accesses requires network communication. Subsequent memory accesses can be performed locally. However, this algorithm is susceptible to *thrashing*: if two processes access the same data item or data items on the same memory page, the data will be transferred back and forth between the two processors. Finding the previous owner of a certain data item to request migration is also an issue. This can be done by broadcasting a migration request to all processors. An alternative solution is maintaining a manager process which keeps track of the data locations.

An important problem in all single-reader/single-writer algorithms is the inherent lack of fault tolerance. Since only one version of each data item exists in the system, if the processor owning this data item crashes, the data is lost. Another problem is that these algorithms can severely limit parallelism as only a single process at a time can access shared data.

When multiple processors need to access the same data at the same time, *replication* can improve the system performance. When data is replicated on multiple hosts, read operations can be performed locally and are therefore very efficient. However, write operations become more expensive. Therefore, replication is a good design choice if the read/write ratio in the application is relatively large.

When one of the replicas is modified, other replicas can be either *invalidated* (i.e. removed) or *updated*. Updating can be done either by *data-shipping*, that is sending the new value of the data item (page, object, depending on the sharing granularity) to all replicas, or by *function shipping*, that is forwarding the *operation* that modifies the data to all replicas and applying this operation on each replica.

The data might be either *fully* or *partially* replicated. In the first case, each processor taking part in the computation has a copy of the data, regardless of whether it ever accesses it or not. In the second case, only part of the processors have a copy of the data. One option is to create a replica on a certain processor when it first accesses a certain shared data item. Another option is to create replicas on processors that frequently read certain data items [31]. Partial replication saves resources – memory needed to store replicas and network bandwidth needed to update/invalidate those replicas. However, it often requires complex administration protocols that keep track of which data is replicated on which processors. This problem becomes particularly difficult in dynamic systems, where processors can join or leave the computation at any time.

The replication protocols come in two basic variants: multiple-readers/single-writer and multiple-readers/multiple-writers. With the single-writer variant, only one process at a time has a write-access to the data. Again, we call this process the *owner* of the data. All write requests must be forwarded to the owner. The owner updates its local copy and invalidates or updates other replicas. This operation must be performed indivisibly. The owner can be either the same process throughout the

whole computation, or the ownership can migrate to a process that wants to perform a write operation. In variants of this protocol, different data items (pages, objects) might have different owners.

The multiple-readers/single-writer replication algorithms have a higher degree of fault tolerance than the single-reader/single-writer approaches. If one replica of the data crashes, the data might still be available at other replicas. However, if the owner of certain data crashes, a special recovery phase is needed before any of the remaining processors can perform write operations on this data.

With the multiple-writers variant, each process might perform write operations on its replica of the data. After updating the local replica, the updates are forwarded to other replicas. This, however, introduces the inconsistency problem: different processors might see different versions of the same shared data. The system must take care that the updates are applied in the proper order. This order depends on the *consistency model* supported by the given Distributed Shared Memory system. An overview of consistency models will be given in the following section.

The multiple-writer replication algorithm also has a higher degree of fault tolerance than the central and migrating manager approaches. However, the possibility of crashes and the dynamic characteristics of the underlying platform introduce a difficult problem: if a processor crashes while performing an update, the update might be forwarded to only a part of the remaining processors which results in inconsistent data. If a processor joins the computation while another processor is updating the data, it may miss the updates performed by this processor. This problem is known as the *atomic multicast problem*. Atomic multicast is non-trivial to implement [165].

5.2.3 Consistency models

A *consistency model* specifies the behavior of the memory subsystem. Ideally, distributed shared memory on a parallel machine should exhibit behavior identical to that of memory on a sequential machine. The consistency model observed by sequential machines is known as *strict consistency* and states that:

> Any read on a data item x returns a value corresponding to the result of the most recent write on x [165].

Implementing strict consistency in distributed systems, however, is impossible due to the lack of absolute global time on which the definition of 'most recent' depends [165]. Therefore, more relaxed consistency models have been designed which provide shared data semantics very close to those of a sequential machine, but are still possible to implement: *sequential consistency* [119] and *linearizability* [99]. Even though possible to implement, those models were still hard to implement *efficiently*, especially in wide-area systems. Therefore, weaker consistency models allowing more efficient implementations have been proposed. In this section, we describe sequential consistency, linearizability and a number of weaker models.

Traditionally, consistency models have been defined in terms of processors operating on memory. In this section, we discuss a different way of specifying memory consistency models: *computation-centric* consistency models. We also discuss

DAG-consistency – a computation-centric model designed specifically for divide-and-conquer applications.

None of the many existing consistency models can meet the needs of *all* applications – consistency requirements are different for different applications or even different data items within one application. The idea of tailoring consistency to the application needs in order to improve performance was first proposed by David Cheriton in 1986 [62]. Since then, many systems with multiple consistency models have been implemented. We provide an overview of such systems in this section.

Finally, we discuss *continuous* consistency models, which allow the programmer to quantify the amount of inconsistency the application can tolerate. These models provide another way of customizing consistency to the application requirements.

Traditional consistency models

The most popular consistency model is *sequential consistency* defined by Lamport in [119]. Sequential consistency states that all processors see the operations on data in *the same* sequential order and the operations by each process appear in this sequence in the order specified by this process' program. Sequential consistency closely resembles the semantics of a sequential data store and is therefore easy to use. It has been implemented in early Distributed Shared Memory systems [123]. However, sequential consistency has a problem of poor performance, especially in wide-area systems.

Linearizability [99] (also known as *atomic consistency*) is stronger than sequential consistency. It assumes that all operations on data receive a timestamp using a global clock with a finite precision (thus not an absolute clock as in strict consistency; a Lamport clock [118] can be used for this purpose). Linearizability extends the conditions of sequential consistency with the requirement that if the timestamp of an operation is smaller than the timestamp of another operation, the former operation should precede the latter operation in the operation sequence seen by the processes. Linearizability is even more expensive to implement than sequential consistency [28].

Causal consistency [104] is based on the notion of *potential causality* introduced by Lamport in [118]. Under causal consistency, all processors must agree on the order of operations that are causally related. Causally unrelated (concurrent) operations can be seen in different orders by different processes. Causal consistency is relatively hard to implement. It requires keeping track of which processes has seen which operations. This can be done using *vector timestamps* [82, 133]. However, vector-timestamp based protocols require large datastructures when large numbers of processors are used. Additionally, support for processors dynamically joining and leaving the computation makes such protocols very complex [105].

Under *PRAM consistency* [127], operations performed by a single process must be seen by all processors in the order they were performed, while operations performed by different processes can be seen in arbitrary order. PRAM consistency can be implemented efficiently in multiprocessor systems because operations can be pipelined (hence the name: PRAM – Pipelined Random Access Memory). However, in dynamic systems the implementation becomes more complex, since special care needs to be

taken that updates are not lost or duplicated when processors are joining or leaving the computation.

Cache consistency [94] (or *coherence* [91]) is a relaxation of sequential consistency. Under cache consistency, operations on each *memory location* have to be sequentially consistent, as opposed to *all operations*. *Processor consistency* [94, 17] is a combination of PRAM and cache consistency: processors might disagree on the order of operations if and only if the operations were performed by different processors and operate on different memory locations. Operations issued by a single processor must be seen in the order imposed by this processor's program. *Slow memory* [127] is a weaker version of PRAM consistency. It requires that operations on a single memory location performed by a single processor are seen by all processors in the same order.

All consistency models described so far enforce a specific order of *individual* operations on the shared data. However, such models might be too restrictive and too inefficient for many applications. *Weak consistency* [73], *release consistency* [91] and *entry consistency* [38] allow the programmer to *group* the operations on the shared data and enforce ordering between the *groups of operations* rather than between individual operations [165]. This is done by introducing *synchronization variables*. Weak consistency introduces one type of operation on synchronization variables: *synchronize(var)*. On invoking this operation, the shared data is synchronized: that is, all local operations performed by the invoking process are propagated to other processes and all operations performed by other processes are applied to the local copy of the invoking process. Accesses to synchronization variables are sequentially consistent. Release consistency distinguishes two types of synchronization operations: *acquire(var)* and *release(var)*. On acquire, all operations performed by local processes are applied to the local copy. On release, local operations are forwarded to other processes. Entry consistency differs from release consistency in that it requires that each shared data item is associated with a synchronization variable. On acquire or release, only data items associated with the synchronization variable are synchronized.

Computation-centric consistency models

While traditional consistency models are *processor-centric*, that is, are expressed in terms of processors operating on a memory, *computation-centric* memory models are expressed in terms of *tasks* (threads) operating on a memory [89]. Computation-centric specification abstracts away the way tasks are mapped to physical processors and is therefore especially suitable for computations is which tasks are dynamically mapped onto available processors. Computation is modeled as a directed acyclic graph (DAG) in which vertices represent tasks and edges represent data-dependencies between tasks.

The computation centric approach makes it possible to express a number of interesting consistency models. One such model is *DAG-consistency* [45] – a consistency model designed especially for divide-and-conquer applications. Under DAG-consistency, tasks may see operations on shared data in different orders but each of those orders must be consistent with the dependencies enforced by the computation DAG. Informally, in divide-and-conquer terms DAG-consistency can be defined re-

cursively in the following way: a task must see all writes its parent must have seen, plus the writes issued by the parent. A task may, but does not have to, see the writes issued by its siblings. A formal definition of DAG-consistency can be found in [45].

DAG-consistency has been implemented in the Cilk divide-and-conquer framework [46] using the Backer algorithm [45] which performs well on a tightly coupled machine like the CM-5, but is not suitable for wide-area systems. Moreover, Cilk's shared memory was developed for pure divide-and-conquer applications which use large data structures, and not for applications that need to share data between sibling tasks (such as branch-and-bound computations). With the Backer algorithm, updates of shared data are passed only along the edges of the execution tree, but not to sibling tasks. Only sibling tasks that execute on the same machine can see each other's updates. Therefore Cilk's shared memory is unsuitable for applications such as branch-and-bound algorithms or game-tree search.

Mixed consistency models

One way of tailoring the consistency criteria to the application needs is proposing *multiple* consistency models to choose from and/or combine. In such systems, the programmer can choose the consistency level on per-application, per-object, per-replica or per-access basis.

Hybrid consistency [88], mixed consistency [16] and Mether [135] allow the programmer to combine two consistency models. Hybrid consistency allows for strong and weak operations. Different levels of consistency can be mixed within one application, but accesses to the same data item must be of the same consistency level. Strong operations appear to be executed in some sequential order. Operations invoked by the same process of which one is strong appear to be executed in the order they were invoked. Agrawal et al. [16] describe mixed consistency in which causal and PRAM memories are combined. In this model, reads are labeled as causal or PRAM. In Mether [135], memory can be accessed in two modes: read-write mode (strongly consistent) and read-only mode (weakly consistent). This is specified when a process maps a shared memory segment into its address space. The programmer can choose to enforce consistency at any point in the program.

The designers of Mermera [98] argue that more levels of consistency are needed in order to better tune the system to the needs of applications. Mermera allows the programmer to choose from four types of memory semantics: sequentially consistent, PRAM, slow and local. Local consistency is a very weak consistency criterion where writes only have to be visible to the process that performed those writes. The consistency level is specified on per access basis: memory writes are labeled with their consistency level. Reads are not labeled, and the semantics of each read is the same: the local copy of an object is returned. Different consistency levels can be mixed within one application and accesses to one object can have different consistency levels. The semantics of such mixed accesses is as follows: sequential writes are totally ordered and this order is consistent with each process' program and with the information flow through weaker writes. PRAM writes and sequential writes satisfy the PRAM order, that is the order consistent with each process' program. Slow, PRAM

and sequential writes satisfy Slow consistency. Local, slow, PRAM and sequential writes satisfy local consistency. Maya [15] also supports four consistency models: sequentially consistent, causal, PRAM and entry consistency [38]. Contrary to other systems, however, Maya does not allow mixing consistency models. The programmer must choose one consistency criterion for the whole application.

GARF [147] is an object-oriented framework which supports five consistency models: slow, PRAM, causal, sequential and linearizability. With GARF, the programmer first describes application functionalities using passive *data objects*. This is done in a centralized and sequential environment. The next step is adapting the application to the distributed environment. Data objects are dynamically bound to *encapsulator objects* which control how data objects send and receive invocations, and *mailer objects* which control the communication between encapsulators. GARF provides a library of encapsulator and mailer objects. Encapsulator objects for handling *asynchrony* (asynchronous invocations), *concurrency control* and *replication* (active and passive) are provided. Among mailer classes provided by GARF, some represent consistency criteria – those are the classes extending the *Mcast* (multicast) class. GARF supports the following types of multicast: slow, PRAM, causal, atomic, sequential and CAtomic (which corresponds to linearizability).

Continuous consistency models

In some systems, consistency requirements are expressed as the maximal allowed distance between the result observed (read) by the application and the ideal result – the result that would be observed with strong consistency (e.g. sequential consistency). This approach is called *continuous consistency* [186], because it explores the continuum between strong consistency, where the difference between the observed and ideal result is zero, and optimistic consistency, where this difference is unbounded.

With N-ignorant transactions [117], the number of updates missed by a replica is bounded – N is a user-defined parameter and an N-ignorant transaction is a transaction that may be ignorant of the results of at most N prior transactions.

With quasi-copies [24] the application programmer can define how much a secondary copy, called a quasi-copy in this context, can diverge from the primary copy. The programmer can choose from three types of consistency conditions: *delay condition* specifies how much time the quasi-copy can lag behind the primary copy. *Version condition* defines how many updates the quasi-copy can miss. This criterion is similar to the N parameter in N-ignorant transactions. *Arithmetic condition* specifies how much the numerical values of the quasi-copy and the primary-copy can differ (for objects with numerical values).

Beehive [158] introduces *delta consistency* – a consistency criterion similar to the delay condition of quasi copies. With delta consistency a read returns a value that was produced at most *delta* time units preceding the read. *Delta* is an application-specified parameter.

Timed consistency [170] requires that if the time of a write is t, the value written by this operation must be visible to all sites in the distributed system by time $t+delta$, where *delta* is an application specific parameter. This criterion is similar to delay

condition of quasi-copies and to delta consistency.

InterWeave [60] supports the notion of a *recent enough* copy. Recent enough comes in six flavours: full coherence – always obtain the most recent version of the object and exclude any concurrent writers, null coherence – always accept the currently cached version, delta coherence guarantees that the object is no more than x versions out of date (similar to N-ignorant transactions and version condition in quasi-copies), temporal coherence guarantees that the object is no more than x time units out of date (similar to delay condition of quasi-copies, delta consistency and timed consistency), finally diff-based coherence guarantees that no more than $x\%$ of the object is out of date.

The most general approach was proposed by Yu and Vadhat in [186], and we describe this approach in more detail here. In their *conit-based continuous consistency model*, applications can define their consistency requirements as *conits* (consistency units). Formally, a conit is a function that maps the shared data state to a real number. Each read *depends* on a number of conits and each write *affects* a number of conits. Each conit has a consistency level, quantified along a three-dimensional vector:

$$\text{Consistency} = (\text{numerical error, order error, staleness})$$

Numerical error is the difference between the observed value of a conit and its ideal value if strong consistency was enforced. *Order error* is the weighted out-of-order writes (i.e., writes that might be rolled back and applied in a different order). *Staleness* is the age of the oldest write affecting the conit that has not been seen by the local replica. For each read, the application can specify the required consistency level of each conit the read depends on. For each write, the application specifies how it affects each conit, that is, how it changes the value of each conit, and what is its order weight with respect to each conit. The conit-based consistency model was implemented in TACT. TACT exports a simple API for defining consistency requirements: the *DependonConit()* function to declare the required consistency level and the *AffectConit()* function to tell the system how a write affects each conit.

Note that, although conits were defined as functions mapping the shared data state to real numbers, the programmer does not need to define such functions. It is enough to specify how each write affects each conit and how each read depends on conits.

The conit-based model elegantly unifies all the models described in this section. Timed consistency can be expressed using the staleness metric. Version and diff-based consistency can be expressed using the numerical error metric. Also, traditional consistency models (e.g., sequential consistency, causal consistency etc.) can be expressed in conit theory. However, it requires conits to be dynamically defined (one conit per access) and the number of conits can be quite large, making the implementation impractical. Moreover, conit-based consistency was not designed with high-performance applications in mind, but applications such as message boards or airline reservation systems. The protocols used in TACT are heavy-weight and less suitable for high performance applications.

5.3 The divide-and-share programming model

To increase the applicability of the Satin framework, we extended the divide-and-conquer model with a shared-data abstraction. We chose a shared objects model since it fits naturally into object-oriented Java and it is possible to implement it efficiently in distributed systems [31]. In the rest of this thesis, we will refer to the divide-and-conquer model extended with shared objects as the *divide-and-share* model.

Shared objects are passed by reference to all or part of the divide-and-conquer tasks. Updates performed on a shared object are visible to all tasks holding a reference to this object. Shared objects are automatically replicated on processors that execute tasks accessing those objects. We use replication on demand: a replica is created on the first access to the object.

Replication is implemented using an update protocol with function shipping: methods that modify the state of the objects are forwarded to other processors, which apply them on their local replicas, other methods are executed only locally. However, distinguishing between the two types of methods is the responsibility of the programmer. The programmer marks part of the methods in the shared object as *shared methods*, and those methods are propagated to other replicas. If a method is not marked as shared, it will not be propagated even if it changes the object state. Automatically distinguishing between local and shared methods is very complex and incurs considerable runtime overhead. Due to Java's support for inheritance, the read-write analysis of methods would have to be performed at runtime (as explained in [130]) which causes performance overhead. Also, many restrictions have to be imposed on the use of shared objects to prevent the programmer from changing the object state in an uncontrolled way. For example, shared object fields cannot be accessed directly (only through methods), shared object methods cannot return an object reference, static fields in shared objects are disallowed, etc. [130].

Because distinguishing between shared and local methods is the responsibility of the programmer, the runtime system cannot guarantee that replicas will remain consistent. However, implementing strong consistency models, such as sequential consistency is not efficient in grid environments anyway. Moreover, many applications do not need strong consistency guarantees. For example, branch-and-bound applications typically do not need any consistency guarantees, as the shared data is used to optimize the search process. Other applications need only very weak guarantees. For such applications, protocols implementing strong consistency would impose an unnecessary performance penalty. Finally, having explicitly inconsistent replicas can be useful for some applications. One example is a replicated transposition table: replicas may contain different numbers of entries depending on the amount of memory available on a processor.

Therefore, Satin's shared objects provide a *user-controlled*, relaxed consistency model called *guard consistency*. Under guard consistency, the user can define the application consistency requirements using *guard functions*. Guard functions are associated with divide-and-conquer tasks. Conceptually, a guard function is executed before each divide-and-conquer task. A guard checks the state of the shared objects

accessed by the task and returns *true* if those objects are in a correct state, or *false* otherwise. Using guards, the programmer can enforce only as much consistency as the application really needs.

Not every consistency criterion can be implemented using guards. The criteria cannot be stronger than DAG-consistency. As mentioned above, under DAG-consistency, a child task must see updates that its parent has seen, as well as updates made by the parent. It may but need not see updates made by its siblings. A guard has exactly the same parameter list as the function implementing the divide-and-conquer task. Therefore, the guard has access to the shared objects used by this task and to the task parameters which depend on the state of the parent that has spawned that task. Therefore the guard can ensure that the state seen by a task is consistent with the state seen by its parent.

The runtime system allows replicas to become inconsistent as long as guards are satisfied: the updates are propagated to remote replicas on a best-effort basis. The runtime system does not guarantee that the updates will not be lost or duplicated. Updates may be applied in a different order on different replicas. This makes using scalable but unreliable broadcasting techniques such as gossiping possible. Also, nodes dynamically joining or leaving the computation are supported. When a guard is not satisfied, the runtime system invalidates the local replicas of shared objects used by the task and fetches a consistent replica from another processor. This will be explained in more detail in section 5.5.

Operations on shared objects are executed *atomically*. The runtime system guarantees that shared object operations do not run concurrently with each other or with divide-and-conquer tasks. An operation performed by a task becomes visible to other tasks only when the system reaches a so-called *safe point*: when a task is creating (spawning) subtasks, when a task is waiting for its subtasks to finish, or when a task completes. Tasks can also explicitly poll for shared object updates. This makes the model clean and easy to use, as the programmer does not need to use locks and semaphores to synchronize access to shared data.

5.4 Programming interface and examples

In this section, we describe the shared objects programming interface and use simple examples to demonstrate how to write parallel applications with the divide-and-share model.

To define a shared object in Satin, the programmer has to write a class that extends the special class *satin.so.SharedObject*. The programmer also needs to use the special interface *satin.so.SharedMethodsInterface* to mark shared methods. This mechanism is similar to the use of the *satin.Spawnable* interface: shared methods must be declared in an interface that extends the empty *satin.so.SharedMethodsInterface*.

Figures 5.3 and 5.4 show an example application that uses shared objects: the Traveling Salesman Problem (TSP). TSP searches for the shortest path through a set of cities. Figure 5.3 shows how the shared objects are declared. TSP uses two shared objects: the *Min* (line 7) object holds the length of the shortest path found so far and

```
 1: interface MinInterface extends satin.so.SharedMethodsInterface {
 2:
 3:   public void set(int val);
 4:
 5: }
 6:
 7: final class Min extends satin.so.SharedObject
 8:   implements MinInterface {
 9:
10:   int val = Integer.MAX_VALUE;
11:
12:   public void set (int new_val) {
13:     if (new_val < val) val = new_val;
14:   }
15:
16:   public int get() {
17:     return val;
18:   }
19:
20: }
21:
22: final class DistTable extends satin.so.SharedObject {
23:   /* ... */
24: }
```

Figure 5.3: Declaring shared objects in the TSP application

the *DistTable* (line 22) object contains a table with distances between each pair of the cities. The *Min* object has two methods: *get()* and *set()*. *Set()* is declared in the *MinInterface* (line 1), which extends the special *satin.so.SharedMethodsInterface* and is therefore a shared method. *Get()* is not declared in this interface and is therefore a local method. The *DistTable* is a constant object – it does not change during the execution. Therefore, all its methods are local (not shown).

Figure 5.4 shows how the shared objects are used in the application. The *tsp()* method (lines 3,10) is a spawnable method, since it is declared in the *TspInterface* (line 1) which extends the *satin.Spawnable* interface. All shared objects accessed by a divide-and-conquer task must be passed to this task as parameters. Therefore *tsp()* has the *Min* and *DistTable* objects in its parameter list. In line 22, the *tsp()* function updates the *Min* object by calling its *set()* method. Since *set()* is a shared method, this invocation will be forwarded to other replicas of the object.

Shared objects are always passed by-reference, unlike 'normal' parameters in Satin which can be passed by-reference or by-value depending on whether the task is executed locally or remotely. When a task is executed remotely, only the shared object reference is transferred to the remote machine, instead of a copy of the object. The task will then access the replica of the object present at the remote machine. If necessary, a new replica will be created.

For each spawnable function, the programmer may define a guard function, in the same class. The name of the guard function is 'guard_<spawnable_function>'. It must have exactly the same parameter list as the spawnable function and return a

```
 1: public interface TspInterface extends satin.Spawnable {
 2:
 3:   public int tsp(int hops, byte[] path,
 4:                 int len, Min min, DistTable dist);
 5:
 6: }
 7:
 8: public class Tsp extends satin.SatinObject implements TspInterface {
 9:
10:   public int tsp(int hops, byte[] path,
11:                 int len, Min min, DistTable dist) {
12:
13:     int[] mins = new int[NRTOWNS];
14:
15:     /*use the shared object to generate a cutoff*/
16:     if (len >= min.get()) {
17:       return len;
18:     }
19:
20:     /*update minimum*/
21:     if (hops == NrTowns) {
22:       min.set(len);
23:       return len;
24:     }
25:
26:     for (int city : getCitiesNotOn(path)) {
27:       /*spawn a new task for each city not on initial path*/
28:       mins[i++] = tsp(hops+1, extendPath(path, city),
29:                   len + dist.getDist(path[path.length-1],city),
30:                   min, dist);
31:     }
32:     sync();
33:
34:     /*return the shortest route*/
35:     return getMinimum(mins);
36:   }
37:
38:   public static void main(String args[]) {
39:
40:     Min min = new Min();
41:     DistTable dist = new DistTable();
42:     Tsp tsp = new Tsp();
43:     int result = tsp.tsp(0, new byte[0], 0, min, dist);
44:     tsp.sync();
45:     System.out.println(``Shortest path:'' + result);
46:
47:   }
48: }
```

Figure 5.4: Using shared objects in the TSP application

boolean value.

Since TSP does not need any consistency guarantees, we use a different application as an example: the Barnes-Hut N-body simulation. This application simulates the behavior of N bodies under influence of forces (e.g., gravitational or electrostatic). The pseudo-code for this application is shown in figures 5.5 and 5.6. The positions of all bodies are stored in a shared object *Bodies*. Figure 5.5 shows the declaration of this object. This object contains the positions and masses of all bodies (*bodyArray*, line 11) and an octagonal tree which represents the space the bodies are in (*bodyTreeRoot*, line12).

Figure 5.6 shows how the shared object is used. The application performs a number of iterations. At the end of each iteration, the root task updates the positions of the bodies and the body tree (figure 5.6, line 52). Before a processor starts executing a task belonging to a certain iteration, it has to make sure that it received the updates belonging to the previous iteration, that is, it checks if its shared object replica is consistent with the replica accessed by the root task. This is done by means of a guard function. The guard function (*guard_ computeForces()*) is shown in figure 5.6, line 35. Its signature is identical to the signature of the spawnable function (*computeForces()*, lines 3,12) except for the return type.

Because shared object invocations are serialized and sent over the network to remote processors, all the parameters of shared methods must be either of basic types or must be serializable. Also shared objects themselves must be serializable, because they are sent to remote processors while creating new replicas. This is, however, ensured by inheriting from the *satin.so.SharedObject* class which is serializable (in Java, all subclasses of a serializable class are serializable as well). The programmer is allowed to use standard Java serialization mechanisms, for example he can provide his own serialization and deserialization methods: *readObject()* and *writeObject()*. Also, the keyword *transient* can be used to declare that certain fields should not be sent over the network. This mechanism can be used to decrease the amount of data sent. For example, in Barnes-Hut, the shared object *Bodies* contains not only the positions of the bodies, but also the body tree. Sending the entire body tree is very expensive, while it can be reproduced using the body positions. Therefore, the programmer can declare the body tree as transient and write a *readObject()* method which creates the body tree after reading the positions of the bodies (figure 5.5, line 26).

5.5 Implementation

We have extended the Satin bytecode rewriter and the Satin runtime system to support shared objects. The bytecode rewriter searches for interfaces extending the special *satin.so.SharedMethodsInterface*. It generates the necessary communication code for all methods found in such interfaces (shared methods).

Unlike the implementation of Java RMI or RepMI [130], we do not use *stubs*: special proxy objects through which all accesses to shared objects must go. Instead, the Satin bytecode rewriter rewrites the shared methods in such a way that before calling the method locally, it is first marshaled (i.e., its identifier and parameters)

```
 1: public interface BodiesInterface
 2:   extends satin.so.SharedMethodsInterface {
 3:
 4:   public void update(LinkedList results, int iteration);
 5:
 6: }
 7:
 8: public class Bodies extends satin.so.SharedObject
 9:   implements BodiesInterface {
10:
11:   Body[] bodyArray;
12:   transient BodyTreeNode bodyTreeRoot;
13:   public int iteration;
14:
15:   /* ... */
16:
17:   public void update(LinkedList results, int iteration) {
18:
19:     this.iteration = iteration;
20:
21:     /*update the body array and the body tree*/
22:
23:   }
24:
25:   /*redefine standard deserialization method*/
26:   private void readObject(java.io.ObjectInputStream in)
27:     throws java.io.IOException, ClassNotFoundException {
28:
29:     /*set all non-transient fields*/
30:     in.defaultReadObject();
31:
32:     /*rebuild the body tree using the bodyArray*/
33:     bodyTreeRoot = buldBodyTree(bodyArray);
34:   }
35: }
```

Figure 5.5: Declaring a shared object in the Barnes-Hut application

```
 1: public interface BarnesHutInterface extends satin.Spawnable {
 2:
 3:   public LinkedList computeForces(byte[] nodeId,
 4:                                    int iteration, Bodies bodies);
 5:
 6: }
 7:
 8: public class BarnesHut extends satin.SatinObject
 9:   implements BarnesHutInterface {
10:
11:   /*spawnable function*/
12:   public LinkedList computeForces(byte[] nodeId,
13:                                    int iteration, Bodies bodies) {
14:
15:     LinkedList res[] = new LinkedList[8];
16:     BodyTreeNode treeNode = bodies.findTreeNode(nodeId);
17:
18:     if (treeNode.children = null) {
19:       /*leaf node, do sequential computation*/
20:       return treeNode.computeForcesSeq(bodies);
21:     } else {
22:       for (int i = 0; i < 8; i++){
23:         if (treeNode.children[i] != null) {
24:           /*spawn child tasks*/
25:           byte[] newNodeId = createNewNodeId(nodeId, i);
26:           res[i] = computeForces(newNodeId, iteration,bodies);
27:         }
28:       }
29:       sync();
30:       return combineResults(res);
31:     }
32:   }
33:
34:   /*guard function*/
35:   public boolean guard_computeForces(byte[] nodeId,
36:                                      int iteration, Bodies bodies) {
37:
38:     return (bodies.iteration+1 == iteration);
39:
40:   }
41:
42:   /*main function, in which the body positions are updated*/
43:   public static void main(String[] args) {
44:     BarnesHut barnesHut = new BarnesHut();
45:     Bodies bodies = new Bodies(NUMBODIES);
46:     for (int iteration = 0; iteration < ITERATIONS; iteration++){
47:       /*spawn*/
48:       LinkedList results = barnesHut.computeForces(rootNodeId,
49:                                                    iteration, bodies);
50:     }
51:     sync();
52:     bodies.update(results, iteration);
53:   }
54: }
```

Figure 5.6: Using a guard function to enforce shared object consistency in Barnes-Hut

and sent to remote replicas. The advantage of this solution is that the programmer can access the fields of a shared object directly which makes the programming model more flexible and easy to use. With stubs, the shared objects could only be accessed through methods. A disadvantage is that shared object references must be handled in a special way. With RMI and RepMI stubs serve as object references. Stubs contain special serialization and deserialization routines which take care that after being sent to a remote machine, the stub points to the right (replica of the) object. Since we do not use stubs, the Satin runtime system must search for shared object references in the data structures sent to remote machines, and ensure that each such reference points to the right replica on the remote machine. This complicates the implementation of the runtime system. Currently, we restrict the way shared object references can be used: shared objects cannot be passed as parameters of shared methods of other shared objects. Shared objects cannot have fields of the *SharedObject* type. Creating data structures (such as arrays, graphs) with shared object references and passing them as parameters to spawnable methods is also forbidden. Shared objects must be included explicitly in the parameter list of a spawnable method. In the future, we want to extend the Satin bytecode rewriter and runtime system to handle also more advanced usages of shared object references.

Replicas of shared objects are created in the following way. If a processor receives a task with a shared object as a parameter, it checks if it has a replica of this object. If it does not have the replica, it copies the object from the machine it received the task from. This way of creating replicas fits the open world model well: a processor can join the computation at any moment and receives up-to-date replicas of all shared objects it needs.

Updates to shared objects are forwarded to remote replicas *asynchronously*. We do not try to prevent updates from getting lost or being duplicated. We do use reliable communication, but since processors can join or leave the computation at any moment, also while a broadcast takes place, a processor can miss an update or receive it twice. The updates may also arrive in a different order at different machines.

Guard consistency is implemented in the following way. Conceptually, a guard function is evaluated for each task (if a guard function is defined). The implementation, however, can make an important optimization. The Satin runtime system only needs to evaluate guards for remote tasks which were obtained from other machines. This approach can be used because the strongest consistency model a divide-and-conquer application may need is DAG-consistency. When a parent and child tasks are executed on the same machine, if a shared object was in a consistent state when the parent was executed, it will also be consistent when the child is executed. Thus, it is sufficient to check the consistency of shared objects for remote tasks.

If a guard evaluates to false, the following actions are taken. First the system waits a certain amount of time for late updates to arrive. If after this time the guard still evaluates to false, the runtime system contacts the processor from which the task was received and requests the replicas of the shared objects used by this task. The machine from which a task was received is the machine on which the parent of this task was executed. So, this machine certainly contains replicas of shared objects that are consistent for this task.

5.6 Divide-and-share applications

In this section, we will describe our experiences with programming grid applications using the divide-and-share model. For each of the applications, we also discuss if it is possible to program it in a pure divide-and-conquer style (i.e., without the shared data abstraction). For the applications that can be implemented without a shared data abstraction, we discuss the benefits of using shared objects. Performance results will be given in section 5.7.

5.6.1 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) application computes the shortest path through a set of cities. Each city should be visited exactly once. We use a branch-and-bound algorithm which recursively searches all possible paths and prunes large parts of the search space by maintaining a global variable containing the length of the shortest path found so far. If the length of a partial path is bigger than the current minimal length, this path is not expanded further and a part of the search space is pruned.

The implementation of TSP in Satin is straightforward (see figures 5.3 and 5.4). A new task is spawned for each partial path. The global minimum is implemented as a shared object. Also the static datastructure containing the distances between all cities is implemented as a shared object to reduce communication overhead. The shared object does not need to be consistent to ensure the correctness of the algorithm. However, delays in update propagation may lead to search overhead.

Implementing TSP in a pure divide-and-conquer style, that is, without a shared data extension, is possible but inefficient, because the possibility of pruning parts of the search space is very limited. Below a certain depth in the search tree, subtrees are evaluated sequentially and within those subtrees sharing of the minimum value and pruning is possible. However, solutions cannot be propagated between those subtrees. This leads to enormous search overhead and slows down the execution of the program by a factor of 100 or even 1000, depending on the problem size and the number of processors used. Using the Younger Brothers Wait Concept (YBWC) [81] can improve the performance. With YBWC, the second and subsequent subproblems are not spawned until the first subproblem is finished. The result returned by the first subproblem is passed to the subsequent subproblems and is likely to cause pruning in those subproblems. This technique reduces the search overhead but also decreases the amount of parallelism and causes load imbalance. Therefore, a pure divide-and-conquer version of TSP with YBWC optimization is still around 40% slower than the divide-and-share version.

5.6.2 LocusRoute

LocusRoute is a VLSI standard cell router. It routes wires between endpoints so as to minimize the total area of the layout. To minimize the area, the algorithm tries to route wires through regions (routing cells) that have few other wires running through them. It calculates a cost function for each route: the number of wires in the routing

cells the route passes, and uses the route with the lowest cost. The total cost of the circuit is the sum of the number of wires running through each routing cell. Because the order of placement of the wires affects the total cost, the program performs a number of iterations. On every iteration except the first one, each wire is 'ripped out' and re-routed. The LocusRoute application is a part of the SPLASH suite [157, 182].

LocusRoute was implemented in Satin by recursively splitting the set of wires into two subsets. The subsets are routed in parallel. A shared object is used for storing the *cost array* - a data structure that keeps track of the number of wires running through each routing cell in the circuit. The data need not be consistent. However, a delay in update propagation may diminish the quality of the resulting circuit.

Implementing LocusRoute in a pure divide-and-conquer style is not possible. Without a shared data abstraction it is not possible to implement the cost array data structure on which the placement of wires depends.

5.6.3 Barnes-Hut N-body simulation

Barnes-Hut simulates the evolution of a large set of bodies under the influence of forces, for example gravitational or electrostatic forces. The evolution of N bodies is simulated in iterations of discrete time steps. If all pairwise interactions between bodies were computed, the complexity of the algorithm would be $O(N^3)$. The Barnes-Hut algorithm reduces this complexity by approximating far away groups of bodies by a single body at the center of the mass of the group of bodies. The precision factor *theta* indicates if a group of bodies is far enough to use this optimization. With a small *theta* the algorithm is faster while with a big *theta* it is more accurate. For the purpose of this optimization, the simulated bodies are organized in a tree structure that represents the space the bodies are in. The root node represents the whole space, its children the subspaces of this space, etc. For each body, the algorithm traverses the body tree. If a body tree node is far away from the given body, all bodies in this node are approximated with a large body in the center of the node and the force is computed. After computing forces for all bodies, the positions of the bodies and the body tree are updated.

In the Satin implementation of the algorithm, a new task is spawned for each node in the body tree. The task calculates forces for all bodies contained in this node. The positions of the bodies and the tree node are stored in a shared object, so that this enormous data structure does not have to be sent over the network each time a task is executed remotely. The shared object is updated at the end of each iteration. The application does have consistency constraints: the updates must be propagated to a processor before it can start working on the next iteration. The consistency of the data is ensured by means of guards, as described above (see figure 5.6).

The Barnes-Hut application can be also implemented in a pure divide-and-conquer style. In that case, the positions of the bodies and the body tree have to be passed as task parameters. This means, however, that the body tree has to be sent over the network each time a task is stolen, which typically is thousands to tens of thousands times during the application run. This would cause significant overhead, as the body tree is a large data structure. The amount of data sent over the network can be

decreased by passing only a *necessary tree* instead of the full body tree as a parameter. A necessary tree contains only those parts of the body tree that are needed for the bodies in the task's part of the tree. However, even with this optimization, the amount of communication in the pure divide-and-conquer version is still larger than in the divide-and-share version.

5.6.4 SAT solver

The satisfiability problem (SAT), that is the problem of deciding whether a given boolean formula is satisfiable, is an important NP-complete problem. The solution of a SAT problem is either a boolean variable assignment that makes the given formula true, or the result 'unsatisfiable' meaning that no such assignment exists. Solving a SAT problem requires a systematic search over a potentially huge solution space. Various techniques have been developed to make this search more efficient for practical problems, but it is inherently difficult. Satisfiability solvers are commonly used in industry to verify the correctness of complex digital circuits, such as out-of-order execution units in modern processors.

The SAT solver used for this thesis is based on SAT4J [6], a reimplementation in Java of MiniSAT [75]. Both MiniSAT and SAT4J are 'industry strength' solvers, that are competitive with other state-of-the-art implementations. The solver uses a backtracking search that speculatively assigns boolean values to variables until the problem is satisfied or a conflict is encountered. Upon a conflict the solver backtracks. Parallelizing SAT4J with Satin was relatively easy. For each speculative assignment a task is spawned so that alternative assignments are evaluated in parallel.

A challenging issue in parallelizing SAT solvers arises from the fact that it is hard to predict how much execution time is needed to solve a spawned subproblem. For some subproblems, the costs of spawning may even exceed the execution time. Therefore, in our implementation we use the approach taken in the GridSAT solver [64]: each task first performs a certain amount of sequential search before splitting up the remaining search problem. This guarantees that only 'hard' tasks will be split.

SAT solvers often implement an iterative strategy to go down the search tree. The purpose of this is to avoid spending too much time in very deep subtrees that might have been cut off more easily if an alternative branch was chosen earlier. In sequential SAT solvers, this can easily be implemented by choosing a certain bound on the total number of assignment conflicts found, and increasing that bound gradually by a certain factor. However, implementing a similar conflict bound with a parallel version is harder, since without communication, a particular branch does not know how many conflicts are generated in other branches, and how the conflicts add up globally. It is possible to make some assumption about the number of conflicts still allowed in a particular subbranch, but this can easily be over- or underestimated, leading either to more iterative restarts, or more searching in fruitless subtrees than the sequential version does. With shared objects, up-to-date knowledge about the global number of conflicts remaining can be obtained almost trivially.

Other aspects that are currently implemented using shared objects are the pruning of subproblems in case a truth assignment is found by one of the searches, and

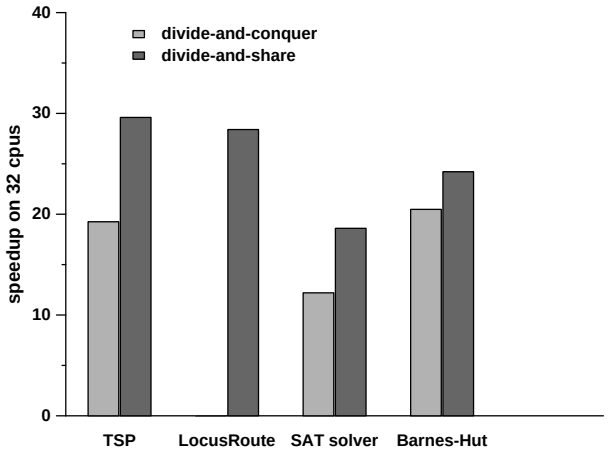


Figure 5.7: Speedups on 32 DAS-2 processors

an implementation of global learning [64]. In global learning, information about conflicting assignments learned locally in one branch of the search tree is made available to other branches in order to potentially cut off related subtrees. As reported elsewhere [43, 64], sharing these learned clauses can indeed potentially help, but also introduces some overhead that has to be earned back. A simple way to decrease the overhead is by restricting the use of global learning to clauses up to a certain length (in general, the shorter the learned clause, the higher its potential impact). It appears that a good maximal length for learned clauses is rather SAT problem dependent; currently we limit it to clauses of up to ten literals. Since knowledge gained by global learning is basically an additional source of information, it does not have to be implemented with strong consistency.

It is possible to implement the SAT solver in a pure divide-and-conquer style. Such an implementation, however, is less efficient. The main reason for this inefficiency is that independent branches cannot share the global number of conflicts found, as described above. Also, global learning is not used in the pure divide-and-conquer version. However, global learning appears to have less influence on the performance of the solver on the SAT problem used by us in this thesis. Finally, instead of using a shared object to notify other branches that a solution has been found and the computation should terminate, the special *abort mechanism* would have to be used [175].

5.7 Performance evaluation

In this section we will evaluate the performance of the shared objects extension. The first part of the evaluation was carried out on the DAS-2 cluster computer (for a description of DAS-2 see section 3.6). To demonstrate that our model is also suitable for

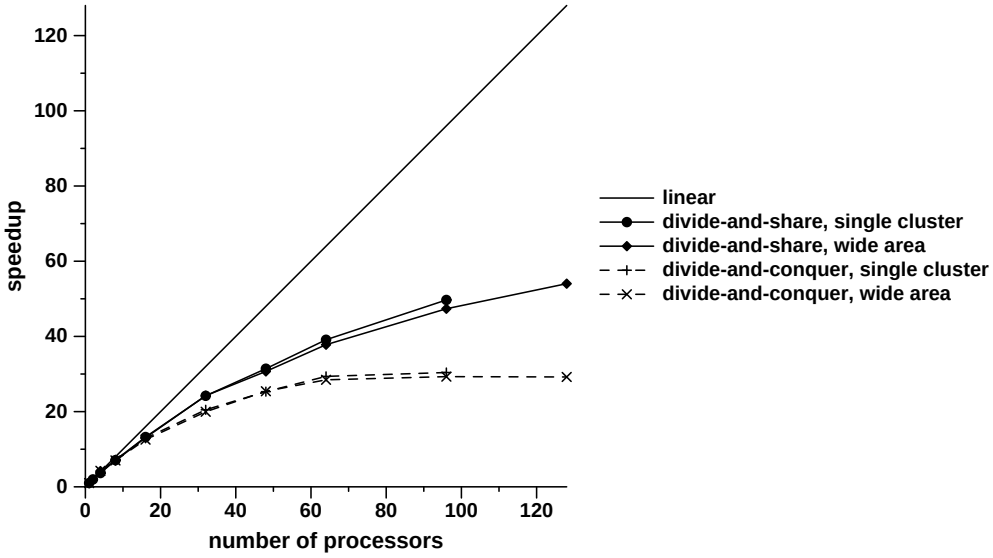


Figure 5.8: Speedups of Barnes-Hut on DAS-2

grid environments, the second part of our experiments is performed on the Grid'5000 testbed [8]. Grid'5000 is a wide-area and heterogeneous system which currently consists of 7 clusters located across France.

In the first part of our experiments, we tested the performance of the applications on a single DAS-2 cluster. For those applications which can be programmed in pure divide-and-conquer style, that is, without shared objects, we compared the performance of the divide-and-conquer version with the performance of the divide-and-share version. We always chose the most efficient divide-and-conquer version, that is, for TSP we chose the Young Brothers Wait version and for Barnes-Hut we chose the Necessary Tree version. We used 32 processors in a single cluster. Figure 5.7 shows the speedups the applications achieved on the DAS-2 cluster. The divide-and-share versions of TSP, SAT solver and Barnes-Hut perform much better than their divide-and-conquer versions. LocusRoute cannot be programmed without shared objects.

For TSP and SAT solver, this performance improvement results from the fact that sharing data allows to diminish the amount of computation. For Barnes-Hut, the performance improvements results from optimizing the communication, which makes the divide-and-share application scale better than the pure divide-and-conquer version. To further study the scalability of both version, we performed an extra experiment with Barnes-Hut. We measured the speedups of both versions on 2 to 128 processors. We tried both a single cluster setting (up 96 processors, because we could not allocate 128 processors on a single cluster) and a wide-area, multi-cluster setting. The results are shown in figure 5.8.

location	processor	cache size
Sophia	AMD Opteron 246 2 GHz	1024 KB
Rennes1	Intel Xeon 2.4 GHz	512 KB
Rennes2	AMD Opteron 250 2.4 GHz	1024 KB
Bordeaux	AMD Opteron 248 2.2 GHz	1024 KB
Orsay	AMD Opteron 246 2 GHz	1024 KB
Lille	Intel Xeon 3.0 GHz	1024 KB

Table 5.1: Processor configurations in the Grid'5000 testbed

	clusters and nr CPUs used	total nr CPUs	normalized nr CPUs
TSP	Sophia 50 Rennes1 40 Bordeaux 30	120	115
LocusRoute	Sophia 50 Rennes1 40 Bordeaux 30	120	94
SAT solver	Orsay 40 Rennes1 32 Bordeaux 40	112	98
Barnes-Hut theta 5.0	Sophia 50 Rennes2 50 Lille 20	120	136
Barnes-Hut theta 7.0	Sophia 50 Rennes2 50 Lille 20	120	126

Table 5.2: Nodes used in the Grid'5000 experiment

The divide-and-share version scales much better that the pure divide-and-conquer version in both single cluster and wide-area, multi-cluster setting.

The second part of the experiments we carried out on the Grid'5000 system. This experiment shows that our model works well in a *real* grid environment. The latency between the clusters used by us ranges from 4 to 10 milliseconds and bandwidth from 200 to 1000 Mbit/s. Grid'5000 is also heterogeneous: it contains machines with different architectures and different speeds. Table 5.1 lists the configuration of the Grid'5000 processors we used.

For each experiment, we use 3 clusters. Table 5.2, lists the clusters and numbers of nodes we used for each experiment.

We compared the performance of our applications in the wide-area, heterogeneous setting with the performance of the same application on a single cluster. To make this comparison meaningful, we need to use the same amount of computational power

in both wide-area and single-cluster experiments. This is not trivial to achieve due to the heterogeneous processor speeds in the grid environment. We used the following methodology. We computed the relative speeds of the processors in each cluster by running a smaller benchmark problem on a single processor in each cluster. We normalized the runtimes of the benchmark problems relative to the runtime on a single processor of the Sophia (for TSP, LocusRoute and Barnes-Hut) or Orsay (for SAT solver) cluster. Next, we computed the normalized number of CPUs. Those numbers are listed in table 5.2. Then, in the single cluster run, we used the same number of processors as the normalized number in the grid runs.

The runtimes and speedups of the applications are listed in table 5.3. TSP and LocusRoute achieve high speedups on the Grid'5000 testbed. The SAT solver performs slightly worse than LocusRoute and TSP. The reason for that is a highly unbalanced search tree which makes it harder to balance the load in the application. Also, the cost of spawning in SAT solver is higher than in the other applications because the whole data structure containing the description of the SAT problem is cloned for each spawned job.

For Barnes-Hut, we experimented with two values of the *theta* constant: 5.0 (which we also used in the DAS-2 experiment) and 7.0. For *theta*=5.0 the speedup is mediocre: 25 which is much smaller than the speedup on a similar number of nodes on DAS-2. This is because the processors in the Grid'5000 testbed are significantly faster than the DAS-2 processors, while the communication speed is similar. Therefore, it is more difficult to achieve high speedups on the Grid'5000 testbed. When *theta*=7.0 the application computes the forces with more accuracy and therefore has a higher computation-to-communication ratio. Thus, the speedup of this version is higher: 89.

For all four applications, the speedups in the wide-area setting were very close to the speedups on a single cluster. This indicates that our algorithms can be run efficiently on wide-area systems even though the applications share significant amounts of data. The amount of data sent by each application is shown in tables 5.4 and 5.5. Column 5 of table 5.4 lists both local-area and wide-area point-to-point traffic. The last column of table 5.5 lists the amount of broadcast traffic.

5.8 Comparison with related work

Few other divide-and-conquer frameworks provide shared data abstractions. Cilk [46] provides a shared memory abstraction for divide-and-conquer computations on the Connection Machine CM-5. Cilk's shared memory implements DAG-consistency using the Backer algorithm [45] which performs well on a tightly coupled machine like the CM-5, but is not suitable for wide-area systems. Moreover, Cilk's shared memory was developed for pure divide-and-conquer applications which use large data structures (such as Barnes-Hut) and not for applications that need to share data between sibling tasks (such as TSP). Updates of shared data are passed only along the edges of the execution tree, but not to sibling tasks. Only sibling tasks that execute on the same machine can see each other's updates. Therefore Cilk's shared memory is unsuitable for applications such as TSP and SAT solver with learned clause sharing.

	runtime Grid	runtime single CPU	grid speedup	single cluster speedup
TSP	200s	5.4h	97	99
LocusRoute	951s	21.5h	81	89
SAT solver	113s	1.7h	54	59
Barnes-Hut (θ 5.0)	226s	1.6h	25	31
Barnes-Hut (θ 7.0)	390s	9.6h	89	89

Table 5.3: Test results the Grid'5000 testbed

	nr tasks executed	nr steal requests	nr tasks stolen	amount of data sent for work stealing local-area/wide-area
TSP	2 000 000	800 000	8 000	33MB/4MB
LocusRoute	160 000	40 000 000	3 000	1.4GB/22MB
SAT solver	160 000	2 000 000	4 000	1.8GB/1GB
Barnes-Hut (θ 5.0)	335 000	5 600 000	18 000	800MB/300MB
Barnes-Hut (θ 7.0)	335 000	6 500 000	20 000	800MB/400MB

Table 5.4: Statistics for Grid'5000 runs

	nr shared object invocations	amount of data broadcast for object updates
TSP	50	5KB
LocusRoute	120 000	23MB
SAT solver	130 000	17MB
Barnes-Hut (θ 5.0)	4	200MB
Barnes-Hut (θ 7.0)	4	200MB

Table 5.5: Statistics for Grid'5000 runs - cont.

Peng et al. [143] noticed this shortcoming of Cilk and implemented SilkRoad – an extension to Cilk that provides global user locks and shared memory with lazy release consistency [114]. SilkRoad was designed to run in a single cluster environment and is not suitable for wide-area grid environments.

Javelin [139] is a framework for writing branch-and-bound applications. Branch-and-bound is similar to divide-and-share but more restrictive. Javelin provides a very limited possibility of sharing data between tasks for bound propagation. All tasks are sharing the current bound, usually an integer or real number, but Javelin allows it to be of any object type. When a task finds a new bound, it broadcasts it to all processors. This is, in fact, replication with data shipping which has been shown to be less efficient than function shipping for object-based shared data models. Javelin does not provide any means of enforcing consistency.

The function shipping approach to object replication in Satin was inspired by Orca [31]. Orca provides sequential consistency which is implemented using totally ordered broadcast. Orca applications achieve good performance on a single cluster, but because of the restrictive consistency model, Orca is less suitable for wide-area systems.

RepMI [130] offers object replication in Java with sequential consistency. The API of RepMI is similar to our API: the programmer uses inheritance and marker interfaces to define replicated objects. The local and shared methods (read and write methods in RepMI's terminology), however, are distinguished automatically by the compiler and runtime system. To prevent the programmer from uncontrolled access to replicated objects, RepMI imposes many restrictions on the programming model, for example, it does not allow direct access to the fields of a shared object. RepMI achieves good performance on a cluster of machines connected with Myrinet [48]. Similar to Orca, however, its restrictive consistency model makes it unsuitable for wide-area computing. Also, read/write analysis, thread scheduling, and indirection in accessing replicated objects adds overhead which is not justified for applications that do not need strong consistency.

5.9 Conclusions

We presented a divide-and-share programming model which combines the divide-and-conquer paradigm with a shared data abstraction – shared objects. The new divide-and-share model has a broader applicability than the pure divide-and-conquer model.

Shared objects implement a new consistency model, guard consistency, designed especially for grid-enabled divide-and-conquer applications. Under guard consistency, the programmer can define the consistency requirements of the application using guard functions associated with divide-and-conquer tasks. A guard function specifies what the status of an object should be for a task to execute correctly. The runtime system allows replicas of shared objects to become inconsistent as long as their guards are satisfied. When a guard is unsatisfied, the system brings the local replica into a consistent state. The guard consistency model is easy to use and allows for efficient

implementation in grid environments. In particular, nodes dynamically joining or leaving the ongoing computation can be tolerated.

We implemented a number of divide-and-share applications using the Satin framework: Locus Route (VLSI routing), SAT solver, Barnes-Hut (N-body simulation) and Traveling Salesman Problem. We evaluated the performance of our applications on the DAS-2 supercomputer and showed that they achieve good speedups on a single cluster. To demonstrate that our model is suitable for *real* Grid environments, we tested it on the wide-area, heterogeneous Grid'5000 testbed and showed that applications using shared data can achieve high speedups in a real grid environment.

Chapter 6

Summary and conclusions

The goal of the research presented in this thesis was simplifying the process of creating grid enabled applications. We proposed that this goal can be achieved by creating a grid programming *framework* – a set of tools that forms a layer of abstraction between the application and the Grid. A framework provides a high-level and easy to use programming model and transparently resolves many grid programming issues.

We started with a prototype divide-and-conquer framework (Satin) designed and implemented by Rob van Nieuwpoort. Thanks to an efficient, grid-aware load-balancing algorithm, the original system could run efficiently over wide-area, heterogeneous systems. However, many issues still needed to be addressed before Satin would become a mature grid programming framework. Those issues we address in this thesis.

In chapter 3, we investigated the problem of fault tolerance, malleability and migration. Grid environments are inherently dynamic, nodes can become available or unavailable at any moment and the application must be able to cope with it. We designed a simple and efficient fault-tolerance algorithm based on recomputing work lost in crashes and restructuring the execution tree to minimize the amount of recomputation. We extended this algorithm to be able to reuse work done by the leaving processors, if they leave gracefully. We also added a simple checkpointing facility that stores intermediate results on a stable storage. This set of algorithms allows divide-and-conquer applications to handle a number of grid scenarios. Applications can tolerate processor crashes and processors dynamically joining and leaving an ongoing computation. Applications can be efficiently migrated or stopped and restarted later on the same or a different set of resources.

In chapter 4, we have investigated the problems of resource selection and adaptive execution. Existing solutions to those problems require that a performance model for an application is known. However, constructing performance models is an inherently difficult task. Therefore, we investigated if it is possible to provide a solution that does not require a performance model. We propose an approach in which an application is started on an arbitrary set of resources. Some simple heuristics can be used to select this initial set (e.g., the fastest available processors), but no advanced models are needed. During the run, we monitor the application performance by collecting

statistics about how much time processors spend communicating or being idle. We use those statistics to deduce the application requirements and adjust the resource set to better fit the application needs. This adjustment is performed by adding or removing nodes to/from the running application. To implement this approach, we added an extra process – an adaptation coordinator, which collects the application statistics and controls adding and removing nodes. We evaluated our approach in a number of scenarios typical for grid environments and we have shown that we can achieve significant performance improvements (10–60% in our experiments).

In chapter 5, we have extended the programming model of our framework. The original Satin framework provided the divide-and-conquer model. A limitation of this model is the lack of a data-sharing abstraction. Therefore, we have extended the divide-and-conquer model with a shared-object abstraction. The API of the shared-object extension is similar to the original Satin API: the programmer uses standard Java mechanisms such as inheritance and marker interfaces to define shared objects and operations on them. The compiler generates the necessary communication code. Therefore, the shared-object model is extremely easy to use.

Implementing a shared-data abstraction in grids is a challenging task due to the distributed and dynamic nature of such environments. Traditional consistency models such as sequential consistency are not suitable for wide-area, dynamic systems. We have designed a novel consistency model, guard consistency, which is suitable for divide-and-conquer applications and allows for efficient implementation in grid environments. Under guard consistency, the programmer defines the application consistency requirements using boolean guard functions associated with divide-and-conquer tasks. The runtime system propagates updates to remote replicas optimistically, that is, without guaranteeing that updates will be applied in a certain order, will not be lost or duplicated. The replicas are allowed to become inconsistent as long as guards are satisfied. When a guard becomes not satisfied, the runtime system brings the local replica into a consistent state.

We implemented a number of applications using shared object abstraction and have shown that it simplifies the programming task, improves application performance and extends the applicability of the Satin framework. We have tested our model both in a single cluster environment and in a wide-area, heterogeneous grid environment and have shown that shared-data applications can achieve high efficiencies in such environments.

The Satin framework that is the result of the work described in this thesis can handle a vast number of scenarios typical for grid environments. Below, we list a number of such scenarios.

- A Satin application can tolerate crashing nodes with minimal loss of work. If the number of crashed nodes is substantial, the adaptation component will attempt to replace the crashed nodes.
- The user can add or remove nodes to a running application. The user can also migrate a running application to a different set of resources.
- The user can stop a Satin application and restart it at a later time on a possibly

different set of resources.

- A Satin application can run in a cycle-stealing environment, that is, expand to new processors if they are idle and release them if another higher-priority application arrives.
- If the user starts a Satin application on an inappropriate set of resources, the resource set will be adjusted. For example, if the initial number of processors is smaller than the application degree of parallelism would allow, the application will automatically expand to more processors. If one of the sites is badly connected, the application will be automatically migrated away from this site.
- If during the application run part of the resources become overloaded (e.g., processors or network links) to an extent that the application performance suffers, the application will be automatically migrated away from the overloaded resources. New resources may be added to replace the removed resources.
- If the application degree of parallelism is changing during the run, the number of processors the application is running on will be automatically adjusted.

The resulting Satin system has also improved applicability. Below, we list application classes that can be programmed using the Satin framework.

- Search and optimization problems, for example the satisfiability problem, the Traveling Salesman Problem, the Knapsack problem, N Queens, etc.
- Astrophysical simulations, for example the Barnes-Hut N-body algorithm [34].
- Grammar based learning [12].
- Parallel rendering (raytracing).
- Bioinformatics computations, for example sequence alignment.
- VLSI routing.
- Game tree searching, for example Othello or Awari.
- Numerical applications, for example matrix multiplication or Fast Fourier Transform.

To summarize, in this thesis we have demonstrated that it is indeed possible to simplify the task of creating grid applications by providing a high-level grid programming framework. The Satin framework that is the result of the work presented in this thesis allows rapid development of grid enabled applications. The programmer expresses the problem at hand in a divide-and-conquer fashion and annotates the sequential code with divide-and-conquer and data-sharing primitives. The Satin bytecode rewriter generates the communication, load-balancing and fault-tolerance code. All grid-related issues are resolved by the framework *transparently* to the application

programmer. Therefore, the application programmer needs to focus his attention only on the problem domain of the application and not on the complexity of the platform the application will be running. We believe that our approach will lead to making the tremendous power of the Grid more accessible and will therefore allow tackling grand computational challenges that could not be solved before.

Bibliography

[1] Einstein@home website. http://einstein.phys.uwm.edu.

[2] Folding@home website. http://folding.stanford.edu.

[3] Java GAT API Description. Online: http://www.cs.vu.nl/ rob/JavaGAT-javadoc/.

[4] LHC@home website. http://lhcathome.cern.ch.

[5] Predictor@home website. http://predictor.scripps.edu.

[6] SAT4J website: http://www.sat4j.org.

[7] SETI@home website. http://setiathome.berkeley.edu.

[8] The Grid'5000 Project. http://www.grid5000.fr.

[9] Distributed Resource Management System (DRMS) User's Guide. Online: http://www.research.ibm.com/drms/api.html, 1995.

[10] Sun Microsystems. Java Remote Method Invocation Specification. Online at http://java.sun.com, 2003.

[11] Unicore plus final report – uniform interface to computing resources. Online: http://www.unicore.org/documents/UNICOREPlus-Final-Report.pdf, 2003.

[12] P. Adriaans and C. Jacobs. Using MDL for grammar induction. In *8th International Colloquim on Gramaticla Inference (ICGI'06)*, Tokyo, Japan, 2006.

[13] H. Afsarmanesh, R. G. Belleman, A. S. Z. Belloum, A. Benabdelkader, J. F. J. van den Brand, G. B. Eijkel, A. Frenkel, C. Garita, D. L. Groep, R. M. A. Heeren, Z. W. Hendrikse, L. O. Hertzberger, J. A. Kaandorp, E. C. Kaletas, V. Korkhov, C. T. A. M. de Laat, P. M. A. Sloot, D. Vasunin, A. Visser, and H. H. Yakali. VLAM-G: A grid-based virtual laboratory. *Scientific Programming*, 10(2):173–181, 2002.

[14] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, 2003.

[15] D. Agrawal, M. Choy, H. V. Leong, and A. K. Singh. Evaluating Weak Memories with Maya. Technical Report TRCS93-23, 30, 1994.

[16] D. Agrawal, M. Choy, H. V. Leong, and A. K. Singh. Mixed Consistency: A Model for Parallel Programming (Extended Abstract). In *Symposium on Principles of Distributed Computing*, pages 101–110, 1994.

[17] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'93)*, pages 251–260, 1993.

[18] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[19] M. Aldinucci, F. Andre, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo. Parallel program/component adaptivity management. In *ParCo 2005*, Malaga, Spain, September 2005.

[20] M. Aldinucci, S. Campa, P. P. Ciullo, M. Coppola, S. Magini, P. Pesciullesio, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In *9th International Euro-Par: Parallel and Distributed Computing*, volume 2790 of *LNCS*, pages 712–721, Klagenfurt, Austria, August 2003. Springer Verlag.

[21] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The cactus worm: Experiments with resource discovery and allocation in a grid environment. *International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[22] G. Allen, W. Benger, T. Goodale, H. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The Cactus Code: A problem solving environment for the grid. In *9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, page 253, Pittsburgh, August 2000.

[23] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schnitke, T. Schutt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Submitted to IEEE.*

[24] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, 1990.

[25] I. Altintas, A. Birnbaum, K. K. Baldridge, W. Sudholt, M. Miller, and C. Amoreira. A framework for the design and reuse of grid workflows. In *1st International Workshop on Scientific Applications of Grid Computing (SAG 2004)*, pages 120–133. Springer-Verlag, LNCS 3458, September 2004.

[26] K. Amin, G. von Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. GridAnt: A client-controllable grid workflow system. In *37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, January 2004.

[27] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' guide to NetSolve V1.4.1. Technical Report ICL-UL-02-05, University of Tennessee, Knoxville, TN, USA, June 2002.

[28] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, May 1994.

[29] R. M. Badia, J. Labarta, R. Sirvent, J. M. Perez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2), 2003.

[30] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer Verlag, January 2006.

[31] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruehl, and M. F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1), February 1998.

[32] J. Baldeschwieler, R. Blumofe, and E. Brewer. ATLAS: An Infrastructure for Global Computing. In *7th ACM SIGOPS European Workshop on System Support for Worldwide Applications*, pages 165–172, Connemara, Ireland, September 1996.

[33] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *9th International Conference on Parallel and Distributed Computing Systems (PCDS-96)*, pages 181–188, Dijon, France, September 1996.

[34] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[35] L. Baudel, F. Baude, and D. Caromel. Object-oriented SPMD. In *5th International Symposium on Cluster Computing and the Grid (CCGrid05)*, Cardiff, UK, May 2005.

[36] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *2nd Symposium on Principles and Practice of Parallel Programming (PPoPP'90)*, pages 168–176, Seattle, WA, USA, March 1990.

[37] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, April 2003.

[38] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, 1991.

[39] R. Bhoedjand, T. Ruhl, R. Hofman, K. Langendoen, H. E. Bal, and M. F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, September 1993.

[40] A. D. Birrel and B. J. Nielson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, February 1984.

[41] R. Bisiani and A. Forin. Multilanguage parallel programming on heterogeneous machines. *IEEE Transactions on Computers*, 37:930–945, August 1998.

[42] R. Bisiani and M. Ravishankar. Plus: a distributed shared-memory system. In *17th Annual International Symposium on Computer Architecture (ISCA'90)*, pages 115–124, New York, NY, USA, 1990. ACM Press.

[43] W. Blochinger, C. Sinz, and W. Küchlin. A Universal Parallel SAT Checking Kernel. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, volume 4, pages 1720–1725, Las Vegas, Nevada, USA, 2003. CSREA Press.

[44] R. Blumofe and P. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, January 1997.

[45] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-Consistent Distributed Shared Memory. In *10th International Parallel Processing Symposium (IPPS '96)*, pages 132–141, Honolulu, Hawaii, April 1996.

[46] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[47] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS'94)*, pages 356–368, November 1994.

[48] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[49] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V: A multiprotocol fault tolerant MPI. *International Journal of High Performance Computing and Applications, to appear*, 2006.

[50] D. M. Breuker. *Memory versus Search in Games.* PhD thesis, Universiteit Maastricht, 1998.

[51] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Dawey. A benchmark suite for high-performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[52] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *4th International Conference on High Performance Computing in Asia-Pacific Region*, Beijing China, 2000.

[53] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: Workflow management for grid computing. In *3rd International Symposium on Cluster Computing and the Grid (CCGrid03)*, pages 198–205, May 2003.

[54] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. NÃĺri, and O. Lodygensky. Computing on large scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Science, to appear*, 2005.

[55] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[56] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and Piranha. *Computer*, 28(1):40–49, January 1995.

[57] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing Workshop*, pages 349–363, 2000.

[58] M. Castro, M. Sequeira, M. Costa, and P. Guedes. Efficient and flexible object sharing. In *International Conference on Parallel Processing*, pages 128–137, Bloomingdale, IL, USA, August 1996.

[59] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *15th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 147–158, 1989.

[60] D. Chen, C. Tang, B. Sanders, S. Dwarkadas, and M. Scott. Exploiting High-level Coherence Information to Optimize Distributed Shared State. In *9th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, San Diego, CA, June 2003.

[61] A. Cherif. *Replication for Fault Tolerant Software Using a Functional and Attribute Grammar Based Computational Model.* PhD thesis, School of Information Science, Japan Advanced Institute of Science and Technology, 1998.

[62] D. R. Cheriton. Preliminary thoughts on problem-oriented shared memory: a decentralized approach to distributed systems. *ACM SIGOPS Operating Systems Review*, 19(4):26–33, 1985.

[63] D.-M. Chiu, M. Kadansky, J. Provino, and J. Wesley. Experiences in programming a traffic shaper. In *5th IEEE Symposium on Computers and Communications (ISCC 2000)*, pages 470–476, 2000.

[64] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *2003 ACM/IEEE conference on Supercomputing (SC '03)*, page 37, Washington, DC, USA, 2003. IEEE Computer Society.

[65] H. Dail, H. Casanova, and F. Berman. A decoupled scheduling approach for the GrADS program development environment. In *2002 ACM/IEEE Conference on Supercomputing (SC'02)*, pages 1–14, Baltimore, Maryland, USA, November 2002.

[66] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[67] G. S. Delp. *The architecture and implementation of MEMNET: a high–speed shared-memory computer communication network*. PhD thesis, University of Delaware, Newark, DE, USA, 1988.

[68] M. den Burger, T. Kielmann, and H. E. Bal. Balanced multicasting: High-throughput communication for grid applications. In *Supercomputing 2005 (SC05)*, page 46, Seattle, USA, November 2005.

[69] S. Djilali, T. Herault, O. Lodygensky, T. Morlier, G. Fedak, and F. Capello. RPC-V: Towards fault-tolerant RPC for internet connected desktop grids with volatile nodes. In *2004 ACM/IEE Supercomputing Conference (SC'04)*, page 39, November 2004.

[70] M. Dobber, G. Koole, and R. van der Mei. Dynamic load balancing experiments in a grid. In *5th International Symposium on Cluster Computing and the Grid (CCGrid05)*, pages 1063–1070, May 2005.

[71] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39:84–90, July 1996.

[72] N. Drost, R. V. van Nieuwport, and H. E. Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *6th International Workshop on Global Peer-2-Peer Computing (GP2P)*, Singapore, May 2005.

[73] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *ACM SIGARCH Computer Architecture News*, pages 434–442, 1986.

[74] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[75] N. Eén and N. Sörensson. An Extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, 2003. Springer.

[76] T. Eickermann, H. Grund, and J. Henrichs. Performance issues of distributed MPI applicatins in a German gigabit testbed. In *6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 3–10. Springer-Verlag, LNCS 1697, 1999.

[77] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[78] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, Balatonfured, Hungary, September 2000. Springer-Verlag, LNCS 1908.

[79] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. S. Jr., and H.-L. Truong. ASKALON: A tool set for cluster and grid computing. *Concurrency and Computation: Practice and Experience*, 17(2–4):143–169, February 2005.

[80] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. In *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, pages 112–123, 1989.

[81] R. Feldmann, P. Mysliwietz, and B. Monien. Game Tree Search on a Massively Parallel System. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess7*, pages 203–218, Maastricht, The Netherlands, 1994. University of Limburg.

[82] C. J. Fidge. Time stamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, 1988.

[83] R. Finkel and U. Manber. DIB – A Distributed Implementation of Backtracking. *ACM Transactions of Programming Languages and Systems*, 9(2):235–256, April 1987.

[84] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. In *12th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 211–223, New York, NY, USA, 1989. ACM Press.

[85] I. Foster. *Designing and Building Parallel Programs*, chapter High Performance Fortran. Addison Wesley, 1995.

[86] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag LNCS 3779, 2005.

[87] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces(TM) Principles, Patterns, and Practice*. June 1999.

[88] R. Friedman. Implementing hybrid consistency with high-level synchronization operations. In *12th Annual ACM symposium on Principles of Distributed Computing (PODC'93)*, pages 229–240. ACM Press, 1993.

[89] M. Frigo and V. Luchango. Computation-centric memory models. In *10th ACM Symposium on Parallel Algorithms and Architectures (SPAA'98)*, pages 240–249, Puerto Vallarta, Mexico, 1998.

[90] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogeneous computing environment. In *5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 180–187. Springer-Verlag, LNCS 1497, 1998.

[91] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Conference on Computer Architecture*, pages 376–387, 1998.

[92] A. Gianelle, M. Sgaravatto, and R. Peluso. DataGrid: Job partitioning and checkpointing, 2003.

[93] E. Godard, S. Setia, and E. L. White. DyRecT: Software support for adaptive parallelism on NOWs. In *15th IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1168–1175. Springer-Verlag, LNCS 1800, 2000.

[94] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, March 1989.

[95] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 43–50, Pittsburgh, Pennsylvania, USA, August 2000.

[96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface. *Parallel Computing*, 22(6):789–828, September 1996.

[97] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35:19–153, 1996.

[98] A. Heddaya and H. Sinha. Distributed Parallel Computing in Mermera: Mixing Noncoherent Shared Memories. Technical Report 1996-005, 7, 1996.

[99] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.

[100] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In *1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)*, pages 214–227, London, UK, 2000. Springer Verlag LNCS 1971.

[101] C. Huang, G. Zheng, S. Kumar, and L. V. Kale. Performance evaluation of Adaptive MPI. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, March 2006.

[102] E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution in grids. *Software – Practice & Experience*, 34(7):631–651, 2004.

[103] F. Huet, D. Caromel, and H. E. Bal. A high performance Java middleware with a real application. In *2004 ACM/IEEE conference on Supercomputing (SC '04)*, Pittsburgh, Pennsylvania, USA, November 2004.

[104] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *10th International Conference on Distributed Computing Systems*, pages 302–311. IEEE Computer Society, 1990.

[105] G. G. R. III. Efficient vector time with dynamic process creation and termination. *Journal of Parallel and Distributed Computing*, 55(1):109–120, 1998.

[106] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot. The implementation of Dynamite: An environment for migrating PVM tasks. *ACM SIGOPS Operating Systems Review*, 34:40–55, July 2000.

[107] C. F. Joerg. *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, MIT Departement of Electrical Engineering and Computer Science, 1996.

[108] D. B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, 1989.

[109] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6:109–133, 1988.

[110] Y. F. K. Aida, W. Natsume. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *3rd International Symposium on Cluster Computing and the Grid (CCGrid03)*, pages 156–163, Tokyo, Japan, May 2003.

[111] L. V. Kale, S. Kumar, and J. DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid02)*, pages 230–237, May 2002.

[112] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[113] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on starndard workstations and operating systems. In *Winter 1994 USENIX Conference*, pages 115–131, 1994.

[114] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[115] T. Kielmann, R. F. Hofman, H. E. Bal, A. Plaat, and R. A. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, Atlanta, Georgia, USA, March 1999.

[116] M. F. K. Kirk L. Johnson and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *15th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 213–228, Copper Mountain Resort, CO, USA, 1995.

[117] N. Krishnakumar and A. J. Bernstein. Bounded ignorance: a technique for increasing concurrency in a replicated system. *ACM Transactions on Database Systems*, 19(4):586–625, 1994.

[118] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[119] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, July 1997.

[120] H.-K. Lee, B. Carpenter, G. Fox, and S. B. Lim. HPJava: Programming Support for High-Performance Grid-Enabled Applications. *International Journal of Parallel Algorithms and Applications (to appear)*.

[121] T. J. Lehman, S. W. McLaughry, and P. Wyckoff. T Spaces: The next wave. In *Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.

[122] W. G. Levelt, M. F. Kaashoek, H. E. Bal, and A. S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software – Practice and Experience*, 22(11):985–1010, 1992.

[123] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *5th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–239, New York, NY, 1986. ACM Press.

[124] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpointing for parallel programs. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'90)*, pages 79–88, March 1990.

[125] K. Li and R. Schaefer. A hypercube shared virtual memory system. In *International Conference on Parallel Processing*, pages 125–131, August 1989.

[126] F. C. H. Lin and R. M. Keller. Distributed Recovery in Applicative Systems. In *1986 International Conference on Parallel Processing*, pages 405–412, University Park, PA, USA, August 1986.

[127] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, September 1988.

[128] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *11th IEEE Symposium on High Performance Distributed Computing (HPDC'02)*, pages 63–72, July 2002.

[129] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGOPS Operating Systems Review*, 11(2):128–137, April 1977.

[130] J. Maassen. *Method Invocation Based Communication Models for Parallel Programming in Java.* PhD thesis, Vrije Universiteit Amsterdam, 2003.

[131] J. Maassen, T. Kielmann, and H. E. Bal. GMI: Flexible and efficient group method invocation for parallel programming. In *6th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR-02)*, Washington,DC,USA, March 2002.

[132] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *Conference on Object Oriented Programming Systems, Languages and Applications*, pages 276–284, San Diego, CA, USA, 1988.

[133] F. Mattern. Virtual time and global states of distributed systems. pages 215–226, 1989.

[134] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington. Workflow enactment in iceni. In *UK e-Science All Hands Meeting*, pages 894–900, September 2004.

[135] R. G. Minnich. *Mether: A Memory System for Network Multiprocessors.* PhD thesis, University of Pennsylvania, 1991.

[136] H. H. Mohamed and D. H. J. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. In *5th IEEE/ACM Symposium on Cluster Computing and the GRID (CCGrid05)*, pages 784–791, May 2005.

[137] M. P. I. F. MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[138] S. Mullender, editor. *Distributed Systems*. Addison Wesley, 1993.

[139] M. O. Neary and P. Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. In *ACM Java Grande/ISCOPE Conference*, pages 56–65, November 2002.

[140] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[141] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatic workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[142] A. Patterson and J. Hennessy. *Computer Organization and Design – The Hardware/Software Interface*. Morgan Kaufmann Publishers, 1998.

[143] L. Peng, W. F. Wong, M. D. Feng, and C. K. Yuen. SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters. In *IEEE International Conferrence on Cluster Computing (Cluster2000)*, pages 243–249, November 2000.

[144] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar. Numerical libraries and the grid: the GrADS experiments with ScaLAPACK. In *2001 ACM/IEEE Conference on Supercomputing (SC'01)*, November 2001.

[145] J. Plank. *Efficient Checkpointing on MIMD architectures*. PhD thesis, Princeton University, 1993.

[146] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology: Systems and Technology*, 4(2):63–79, June 1996.

[147] K. R. M. Rachid Guerraoui, Benoit Garbinato. The GARF Library Of DSM Consistency Models. In *6th ACM SIGOPS European Workshop*, pages 51–56, 1994.

[148] U. Ramachandran and M. Y. A. Khalidi. An implementation of distributed shared memory. *Software – Practice and Experience*, 21(5):443–464, May 1991.

[149] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[150] D. A. Reed, C. L. Mendes, and C. da Lu. *The Grid: Bluepring for a New Computing Infrastructure (Second Edition)*, chapter Application Tuning and Adaptation. Morgan Kaufmann Publishers, 2004.

[151] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *7th IEEE Symposium on High-Performance Distributed Computing (HPDC'98)*, pages 172–179, Chicago, IL, USA, July 1998.

[152] J. Robinson, S. Russ, B. Heckel, and B. Flachs. A task migration implementation of the Message-Passing Interface. In *5th IEEE International Symposium on High Performance Distributed Computing (HPDC'96)*, pages 61–68, Syracuse, NY, USA, August 1996. IEEE Computer Society.

[153] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *1st USENIX Symposium on Operating Systems Design and Implementation*, pages 101–114, November 1994.

[154] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for grid computing. In *3rd International Workshop on Grid Computing (GRID 2002)*, pages 274–278, Baltimore, MD, USA, November 2002. Springer Verlag, LCNS 2536.

[155] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *Heterogeneous Computing Workshop*, pages 3–16, 2000.

[156] E. H. Siegel and E. C. Cooper. Implementing distributed Linda in Standard ML. Technical Report CMU-CS-91-151, Carnegie Mellon University.

[157] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Computer Architecture News*, 20(1):5–44, 1992.

[158] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in Beehive. In *9th Annual ACM symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 211–220. ACM Press, 1997.

[159] H. Soh, S. Haque, W. Liao, and R. Buyya. *Advanced Parallel and Distributed Computing*, chapter Grid Programming Models and Environments. Nova Science Publishers, 2006.

[160] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th International Parallel Processing Symposium*, pages 526–531. IEEE Computer Society, 1996.

[161] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23:54–64, May 1990.

[162] V. S. Sunderam. PVM:a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[163] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *3rd International Conference on Logic Programming*, pages 84–98, London, UK, July 1986.

[164] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

[165] A. S. Tanenbaum and M. van Steen. *Distributed Systems, Principles and Paradigms*. Prentice Hall, 2002.

[166] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job schedurer. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[167] K. Taura, K. Kaneda, T. Endo, and A. Yonezawa. Phoenix: A parallel programming model for accommodating dynamically joining/leaving resources. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 216–229, October 2003.

[168] I. Taylor, M. Shields, and I. Wang. Resource management for the Triana peer-to-peer services. In J. Nabrzyski, J. M. Schopf, and J. Weglarz, editors, *Grid Resource Management*, pages 451–462. Kluwer Academic Publisher, 2004.

[169] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323–356, February–April 2005.

[170] F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *18Th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, pages 163–172. ACM Press, 1999.

[171] S. S. Vadhiyar and J. J. Dongarra. SRS:a framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[172] S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in Grid computing. *Concurrency and Computation: Practice and Experience*, 17(2–4):235–257, 2005.

[173] S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2–4):235–257, 2005.

[174] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8):942–947, August 1997.

[175] R. V. van Nieuwpoort. *Efficient Java-Centric Grid Computing*. PhD thesis, Vrije Universiteit Amsterdam, 2003.

[176] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *8th ACM SIGPLAN symposium on Principles and practices of parallel programming (PPoPP '01)*, pages 34–43, New York, NY, USA, 2001. ACM Press.

[177] R. V. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel computing in Java. In *ACM Java Grande Conference*, pages 8–14, San Francisco, CA, USA, June 1999.

[178] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005.

[179] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7–8):1079–1107, June 2005.

[180] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):643–662, 2001.

[181] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, October 1999.

[182] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 24–36, New York, NY, USA, 1995. ACM Press.

[183] N. Wooy, S. Choi, H. Jung, J. Moon, H. Y. Yeom, T. Park, and H. Park. MPICH-GF: Providing fault tolerance on grid environments. In *3rd IEEE/ACM Symposium on Cluster Computing and the Grid (CCGrid03)*, May 2003.

[184] I.-C. Wu and H. T. Kung. Communication complexity for parallel divide-and-conquer. In *32nd Annual Symposium on Foundations of Computer Science*, pages 151–162, San Juan, Puerto Rico, 1991.

[185] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, September 1974.

[186] H. Yu and A. Vadhat. Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *ACM Transactions on Computer Systems*, 20(3):239–282, 2002.

[187] J. Yu and R. Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *5th IEEE/ACM International Workshop on Grid Computing (Grid 2004)*, pages 119–128, November 2004.

[188] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3–4):171–200, September 2005.

Complexiteit en verandering in grid computing

Het doel van *grid computing* is het aan elkaar koppelen en integreren van verschillende computersystemen zodat ze gebruikt kunnen worden als een virtuele supercomputer, die we een *grid* of een *gridomgeving* noemen. De rekenkracht van zo'n virtuele supercomputer is vele malen groter dan de rekenkracht van een traditionele parallelle computer. Een grid kan daarom gebruikt worden om uitermate ingewikkelde problemen op te lossen die niet opgelost zouden kunnen worden door een traditionele supercomputer.

De complexiteit van gridomgevingen is echter ook vele malen groter dan de complexiteit van traditonele supercomputers. In de eerste plaats zijn gridomgevingen *heterogeen*, dat wil zeggen dat ze uit systemen bestaan die mogelijk verschillend zijn voor wat betreft processoren en besturingssytemen. De verschillen in processorsnelheden kunnen enorm zijn. Ook de kwaliteit van netwerkverbindingen varieert van snelle LAN netwerken tot langzame WAN netwerken.

In de tweede plaats zijn gridomgevingen *dynamisch*. Ze bestaan uit grote hoeveelheden computers en daarom is de kans dat sommige computers uitvallen groot. Een gedeelte van de door een applicatie gebruikte computers kan ook overgenomen worden door een andere applicatie met een hogere prioriteit. Ook varieert de belasting op computers en netwerken continu.

Daarom is het schrijven van gridapplicaties een uitermate ingewikkelde taak. De programmeur moet een goed begrip hebben van niet alleen het applicatiedomein maar ook van complexe problemen van het domein van parallel programmeren, zoals het optimaliseren van de communicatie tussen de processoren, foutbestendigheid, adaptiviteit, enzovoort.

In deze dissertatie kijken we naar mogelijkheden om het schrijven van gridapplicaties te vergemakkelijken. We geloven dat dit doel bereikt kan worden door gebruik van *gridprogrammeeromgevingen*. Een gridprogrammeeromgeving is een verzameling programma's, zoals compilers en bibliotheken, die bepaalde taken van de gridprogrammeur overnemen, bijvoorbeeld het verdelen van taken tussen processoren, het optimaliseren van de communicatie of het foutbestendig maken van de applicatie.

De gridprogrammeeromgeving die beschreven wordt in dit proefschrift spitst zich toe op een bepaalde klasse van applicaties, namelijk *verdeel-en-heersapplicaties*. Ap-

plicaties van die soort splitsen een probleem op in deelproblemen, totdat het werk zover opgesplitst is dat het eenvoudig uitgevoerd kan worden. Tenslotte worden alle deeloplossingen gecombineerd tot het uiteidelijke resultaat. Verdeel-en-heersapplicaties kunnen efficiënt worden uitgevoerd op parallelle computers door verschillende taken (deelproblemen) te laten berekenen door verschillende processoren.

Onze programmeeromgeving heet *Satin* en is gebaseerd op een prototype dat is ontwikkeld door Rob van Nieuwpoort. Van Nieuwpoorts prototype implementeert een efficiënt taakverdelingsalgoritme: *Cluster-aware Random Work Stealing* (CRS). CRS is gebaseerd op het *stelen* (overnemen) van taken van willekeurige machines in het systeem. Dankzij dit algoritme kunnen Satin-applicaties erg efficiënt draaien in omgevingen met langzame WAN netwerken.

In hoofdstuk 2 van dit proefschrift beschrijven en classificieren we de bestaande gridprogrammeeromgevingen en beschrijven we het Satin prototype. We leggen uit wat er nog moet gebeuren om dit prototype om te zetten in een volwaardige gridprogrammeeromgeving.

In hoofstuk 3 onderzoeken we hoe we verdeel-en-heersapplicaties *foutbestendig* en *malleable* kunnen maken. We zeggen dat een applicatie *foutbestendig* is als zij uitvallende processoren kan tolereren. We noemen een applicatie *malleable* als ze op een steeds veranderende verzameling processoren kan draaien, dat wil zeggen, dat processoren kunnen komen en gaan terwijl de applicatie draait. Beide eigenschappen zijn belangrijk voor applicaties die in dynamische gridomgevingen draaien. We beschrijven een verzameling algoritmes die verdeel-en-heersapplicaties foutbestendig en malleable maken. De basis van die algoritmes is steeds dezelfde: de resultaten die verloren gingen door het wegvallen van een processor worden herberekend. Echter, om de hoeveelheid werk dat herberekend moet worden te minimalizeren, gebruiken we verschillende technieken:

- We gebruiken de resultaten van zogenaamde *weestaken* opnieuw. Weestaken zijn taken (deelproblemen) die werden gestolen van weggevallen processoren.
- Als we weten dat sommige processoren binnenkort niet meer ter beschikking van de gridapplicatie zullen staan, slaan we de door die processoren berekende resultaten op, om ze later opnieuw te kunnen gebruiken.
- We slaan regelmatig de resultaten van deelproblemen op op een vaste schijf. Als een processor wegvalt, kunnen wij de resultaten van deze processor terughalen van de harde schijf en ze hergebruiken.

Dankzij deze algoritmen kan een applicatie verschillende situaties overleven die kenmerkend zijn voor een grid omgeving:

- Een applicatie kan blijven draaien ondanks wegvallende processoren.
- Processoren kunnen worden toegevoegd aan of weggehaald van een draaiende applicatie.
- Een applicatie kan gemigreerd worden naar een andere verzameling processoren.

- Een applicatie kan worden gestopt en later opnieuw gestart op een verschillende verzameling processoren.

In hoofdstuk 4 kijken we naar het probleem van *selectie van processoren* en *adaptiviteit*. Om efficiënt te kunnen draaien heeft een applicatie een juiste verzameling processoren nodig. De hoeveelheid processoren moest juist zijn, de processoren moeten niet te langzaam zijn en de netwerkverbindingen tussen processoren moeten voldoende snel zijn. De vereisten verschillen per applicatie. Vaak moet de verzameling processoren worden aangepast tijdens de berekening omdat de belasting van processoren en netwerkverbindingen kan veranderen, een gedeelte van de processoren kan wegvallen, of sommige fasen van de aplicatie meer rekenkracht nodig hebben.

Van oudsher werden deze problemen opgelost door middel van een *performance model*. Een dergelijk model is een wiskundige formule die gebruikt wordt om te berekenen hoe snel een applicatie zou draaien op een gegeven verzameling processoren. Om de optimale verzameling processoren te selecteren worden verschillende verzamelingen geëvalueerd door middel van een performance model. De verzameling waarop de applicatie het snelst zou draaien wordt gekozen. Tijdens de berekening wordt de verzameling processoren herhaaldelijk opnieuw geëvalueerd. Als er een betere verzameling processoren is gevonden wordt de applicatie daarnaar gemigreerd, waardoor zij zich kan aanpassen aan de veranderingen in de gridomgeving waarin ze draait.

Het vinden van een performance model voor een applicatie is echter uitermate gecompliceerd. Daarom wordt in dit hoofdstuk een alternative benadering gepresenteerd. In deze benadering wordt een applicatie op een willekeurige verzameling processoren gestart. Terwijl de applicatie loopt worden statistische gegevens verzameld, onder meer over de mate waarin de applicatie een beroep doet op de processor of het netwerk. Deze gegevens worden gebruikt om af te leiden hoe de verzameling processoren aangepast kan worden om de applicatie efficiënter te laten draaien. We laten zien dat we met deze benadering:

- De verzameling processoren automatisch kunnen aanpassen aan de behoeften van de applicatie. Als sommige fasen van de applicatie bijvoorbeeld meer rekenkracht nodig hebben, wordt de verzameling processoren automatisch uitgebreid.
- De applicatie automatisch kunnen migreren van een zwaar belaste verzameling processoren naar een andere, mogelijk minder zwaar belaste, verzameling.
- Processoren met langzame netwerkverbindingen kunnen laten weghalen.
- Nieuwe processoren kunnen toevoegen als een gedeelte van de processoren wegvalt.

In hoofdstuk 5 kijken we naar het programmeermodel van onze omgeving. Een belangrijk nadeel van het verdeel-en-heersmodel is het ontbreken van globale variabelen. Daarom breiden we het verdeel-en-heersmodel uit met globale objecten die door alle taken gelezen en geschreven kunnen worden. Een belangrijk probleem bij het implementeren van zulke globale objecten is de *consistency*. Traditionele

consistency-modellen zijn moeilijk efficiënt te implementeren in gridomgevingen omdat netwerkverbindingen traag zijn en omdat de verzameling processoren waarop de applicatie draait kan veranderen. Daarom hebben wij een nieuw consistency-model ontwikkeld onder de naam *guard consistency*. In dit model definieert de programmeur wanneer de objecten consistent zijn door middel van booleaanse *guard functions*. De programmeeromgeving zorgt er niet voor dat alle kopieën van een globaal object identiek zijn. Het zorgt er alleen voor dat de guardfuncties altijd *true* retourneren. In dit hoofdstuk worden globale objecten gebruikt om een aantal nieuwe applicaties te implementeren om aan te tonen dat deze applicaties efficiënt kunnen draaien in gridomgevingen.

Het resultaat van deze dissertatie is een programmeeromgeving die het eenvoudiger maakt om gridapplicaties te schrijven. Grid-gerelateerde problemen zoals communicatie, taakverdeling tussen processoren of foutbestendigheid worden automatisch opgelost door de programmeeromgeving. De applicaties die geïmplementeerd worden binnen deze programmeeromgeving kunnen efficiënt draaien in gridomgevingen en ze zijn bestand tegen situaties die kenmerkend zijn voor dat soort omgevingen, zoals wegvallende processoren of veranderende belasting op processoren en netwerk verbindingen.

Publications

1. Gosia Wrzesińska, Jason Maassen and Henri E. Bal. Self-Adaptive Applications on the Grid. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07), pp. 121–129, San Jose, CA, USA, 14-17 March 2007.*

2. Gosia Wrzesińska, Jason Maassen, Kees Verstoep and Henri E. Bal. Satin++: Divide-and-Share on the Grid. *2nd IEEE International Conference on e-Science and Grid Computing, Amsterdam, The Netherlands, 4-6 December 2006.*

3. Gosia Wrzesińska, Rob V. van Nieuwpoort and Henri E. Bal. Fault-tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), 4–8 April 2005, Denver, CO, USA.*

4. Gosia Wrzesińska, Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. *International Journal of High Performance Applications, Vol. 20, No. 1, pp. 103–114, Spring 2006.*

5. Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, Thilo Kielmann, and Henri E. Bal. Adaptive Load Balancing for Divide-and-Conquer Grid Applications. *Journal of Supercomputing, 2006.*

6. Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment *Concurrency & Computation: Practice & Experience, Vol. 17, No. 7–8, pp. 1079–1107, June–July 2005.*