Efficient Java-Centric Grid-Computing

VRIJE UNIVERSITEIT

Efficient Java-Centric Grid-Computing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 4 september 2003 om 13.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

Robert Vincent van Nieuwpoort

geboren te Alphen aan den Rijn

promotor: prof.dr.ir. H.E. Bal

Slainte Mhath!

Contents

Contents

Contents

List of Figures

List of Tables

Acknowledgments

It is not titles that honor men, but men that honor titles.

- Niccolo Machiavelli

I consider myself extremely lucky that I had the opportunity to be a Ph.D. student in Henri Bal's group. I think Henri has gathered a group of people around him that is doing good research. But, equally important, they are all, without exception, nice people! There is some form of "chemistry" at work between the individuals in the group that made (and makes!) it a stimulating environment for me to do my research.

Henri takes the task of supervising his Ph.D. students very seriously. We discussed my progress once per week, and he guided me in the right direction with many good suggestions. Still, he left plenty of room for me to follow my own research interests. He always gave only suggestions, never orders. The living proof of this is Ibis, which started as a "pet project" of Jason and myself. Henri also taught me about writing scientific papers and about doing research in general. He also read this document several times, and corrected many of my mistakes. On both the professional level and the personal level, I hold Henri in high esteem. Thanks for your support!

During my first year as a Ph.D. student, Aske Plaat was my second supervisor. He had many good suggestions on the first RMI papers. Aske taught me what research is, and what is not. But, most importantly, he pointed me in the direction of divide-and-conquer parallelism. I really enjoyed working in that field. When Aske left our group, Thilo Kielmann became my second supervisor. He also read this thesis and gave many useful comments. After three and a half years, he even became my "boss", as he hired me as postdoc for the GridLab project. This way, Thilo gave me the opportunity to spend some time on my thesis. We also had lots of fun in Italy, Poland and Hungary. Both Aske and Thilo had a profound influence on my research.

The best memories I have of my Ph.D. time are those of Jason Maassen and Ronald "Ronnie" Veldema. We became friends when we started at the VU as students. We did many practical courses together, and in our spare time also hacked game engines in assembly. We spent hours and hours on those crappy machines in the PC room on the fourth floor. We even had a "hackfest" where we were programming and playing Quake till we dropped. Jason and I eventually wrote our Master's thesis together. After graduating, we all started as Ph.D. students in Henri's group, and worked closely together on the Manta project. We were all in the same room and had truly enormous amounts of fun during that time. But, it was not only fun, we also had many (sometimes heated)

scientific discussions. I think all three of us influenced the work each one of us has done to a great extent. We even went on a holiday together after presenting the Manta RMI paper at PPoPP in Atlanta. We rented a car and drove around Florida for three weeks, and Ronnie went out hunting for a jungle croc (remember that song, Ronald?). Now that Ronald has left our group (and went to work for the competition!), I really miss making sarcastic jokes back and forth with him. Luckily, Jason remained at the VU, and we still work closely together on Ibis and GridLab. By the way, Jason, you won in the sense that you were the first to finish your thesis, but I can beat you with Quake anytime. (Remember the LAN party at the ASCI conference?) You have to get your priorities straight dude!

There are three excellent programmers working in Henri's group: Rutger Hofman, Ceriel Jacobs and Kees Verstoep. They are an important piece of the puzzle that makes our group work. When I finished my computer science study, I thought I was a good programmer. Boy was I wrong! Rutger taught me a lot about threads, synchronization, and upcalls. Ceriel also taught me a thing or two. For example, I remember that I had once declared a variable inside the Manta runtime system called "log". In some other part of the runtime system, Ceriel was trying to call the function "log", but, because of the naming clash, was jumping to the address of my variable instead. At that moment, it became clear to me that programming in C requires more discipline than I had previously thought. Kees did much work behind the scenes. He maintained LFC, and he made sure that all our hardware and software kept running. He did this so well that we actually hardly ever had to ask him for something. Everything just always worked. Manta was a rather large project, and we could never have gotten it to work without the three programmers. A big thank you!

I also would like to thank the members of the thesis committee: Dick Epema, Sergei Gorlatch, Aske Plaat, John Romein and Rich Wolski. They reviewed an earlier version of this document, and gave very useful comments on it. I had many fruitful discussions with John, especially about Satin's abort mechanism. John also gave lots and lots of extremely detailed remarks and good suggestions on this book. His comments basically hit the nail on the head most of the time. The drawback of this was that it took quite some work to process all remarks! However, I was able to improve this document thanks to his comments, so I greatly appreciate his help. Raoul Bhoedjang gave useful feedback on the first RMI work. Lionel Eyraud did some work to make Panda's cluster simulator more flexible. Maik Nijhuis and Martijn Bot proofread parts of this document.

Being a Ph.D. student and especially writing a thesis is not only fun and games of course. I often ventilated my frustrations on my friends and family. So I would like to thank all my friends and especially my family for keeping up with me. My parents gave me the opportunity to study, and always stimulated me to go on. My brother, Frans, allowed me to get rid of my aggression by being cannon fodder in many games of Quake. He also insisted that he was to be mentioned in the acknowledgments, so here you are!

Finally, I think my girlfriend, Patricia, has suffered the most. She had to listen to all my complaints and incomprehensible techno-babble all the time. She never even complained. Instead, she encouraged me and just took over all chores when I was busy working on this thesis *again*. I love you too.

Rob

About the Cover

Niccolo Machiavelli

Niccolo Machiavelli (May 3, 1469, Florence-June 21, 1527, Florence) was an Italian writer and statesman, Florentine patriot, and original political theorist whose principal work, The Prince, brought him a reputation of amoral cynicism.

The phrase *divide et impera*, which means *divide and rule* (divide and conquer) in English, was the motto of Niccolo Machiavelli. It was also used by Philip of Macedon, Louis XI of France and of Austria, Polybius, Bossuet, and Montesquieu. The divide-and-conquer programming model is one of the themes in the thesis.

The cover of this thesis features a scan of a letter that Machiavelli wrote. Machiavelli was secretary of the "Council of Ten of Liberty and Peace in Florence" during the French dominance over Florence. His main duty was to insure the dominance of Florence over neighboring Pisa. In this letter he announces that he is busy laying siege to Pisa and that during the fighting, Pisa will be off limits to Florentines. The letter was written by a secretary but the annotations are by the hand of Machiavelli.

The Recursive Spheres Figure

The cover also features a figure that contains a number of spheres. The idea is that this is a *recursive* figure: the large sphere in the center is surrounded by smaller spheres, which are in turn surrounded by again smaller spheres, etc. Recursion is inherent to the divide-and-conquer programming model. The image is raytraced by an application program written by Ronald Veldema. In this thesis, we use a parallel version of this raytracer that generates this figure for performance benchmarks.

Chapter 1

Introduction

```
There is nothing more difficult to take in hand, more perilous
to conduct, or more uncertain in its success, than to take the
lead in the introduction of a new order of things. Because the
innovator has for enemies all those who have done well under the
old conditions, and lukewarm (indifferent, uninterested)
defenders in those who may do well under the new.

- Niccolo Machiavelli
```

1.1   Motivation

Grid computing is an interesting research area that integrates geographically-distributed computing resources into a single powerful system. Many applications can benefit from such an integration [57, 154]. Examples are collaborative applications, remote visualization and the remote use of scientific instruments. Grid software supports such applications by addressing issues like resource allocation, fault tolerance, security, and heterogeneity. Parallel computing on geographically distributed resources, often called distributed supercomputing, is one important class of grid computing applications. Projects such as SETI@home [156], Intel's Philanthropic Peer-to-Peer Program for curing cancer [78] and companies such as Entropia [50] show that distributed supercomputing is both useful and feasible. However, there are still several difficult problems that have to be solved to make grid computing available for a larger class of users and applications. This thesis attempts to solve some of these problems.

The ultimate goal of the research in this thesis is *to provide an easy to use programming environment for medium and fine grained distributed supercomputing on hierarchical heterogeneous grids*. We will explain our goals in more detail in the following sections. We outline the problems that have to be solved to reach our goals, and present our solutions. A summary of the problems we encountered, and the solutions we devised is presented in Table 1.1.

problem	solution	chapter
heterogeneity of grid environments (Sec. 1.1.1)	Java-centric system	2
Java sequential performance	JITs, native compilers	2
Java communication performance	- efficient serialization	3, 7
	- user-level communication	3, 7
	- zero-copy protocol	3, 7
	- streaming of data	3, 7
fine grained distributed supercomputing (Sec. 1.1.2)	exploit hierarchical structure	4, 6
grid programming models	- RMI & wide-area optimizations	3, 4
	- divide-and-conquer	5, 6, 8
support for speculative parallelism	inlets, exceptions and aborts	8

Table 1.1: Problems that are discussed in this thesis and their solutions.

1.1.1   Heterogeneity of Grid Environments

Due to the heterogeneous nature of the grid, it is difficult to run binary applications. Problems that arise are, for instance, differences in operating systems, libraries, instruction sets, byte ordering, word length and floating point semantics. Currently, researchers either select a homogeneous set of machines from the available pool of computing resources to run a single binary program on, or they use a limited set of binaries, compiled for different architectures and operating systems, to provide limited support for heterogeneity. When the number of different architectures is as large as in a grid environment, providing binaries for all architectures is a cumbersome task.

Most grid computing systems are language neutral and support a variety of programming languages. Recently, interest has also arisen in grid computing architectures that are centered around a single language. This approach admittedly is restrictive for some applications, but also has many advantages, such as a simpler design and the availability of a single type system. In [171], the advantages of a Java-centric approach to grid computing are described, including support for code mobility, distributed polymorphism, distributed garbage collection, and security. Java is sometimes also used as a glue language to allow inter operability between different systems [136].

Java's "write once, run everywhere" feature effectively solves the heterogeneity problems. Therefore, the use of Java for grid computing allows researchers to focus on more interesting problems, such as new grid programming models, load balancing and fault tolerance. However, when Java is to be used for distributed supercomputing, two important problems have to be solved. First, as performance is of critical importance for distributed supercomputing, Java's sequential execution speed should be comparable to compiled C or Fortran applications. We believe that this problem is at least partially solved with the recent Java JIT (Just In Time) compilers, which achieve excellent performance. We expect more improvements in the future. (See also [29, 31, 92, 119, 135, 141].) Second, because distributed supercomputing applications depend on efficient communication, Java's performance problems in this area have to be solved. Particularly, Java's Remote Method Invocation mechanism (RMI) [157] is known to be slow. The first part of this thesis provides solutions for Java's communication performance problems. We will describe an efficient, lightweight RMI implementation that achieves a performance close

to that of C-based Remote Procedure Call (RPC) protocols. Other forms of object-based communication are also investigated.

1.1.2 Distributed Supercomputing on Hierarchical Systems

The aforementioned distributed supercomputing projects only work well because they are extremely coarse grained. In this thesis we will look at how to develop medium and fine grained applications for grids. The communication requirements of the finer grained applications introduce a new problem. In distributed supercomputing, platforms are often hierarchically structured. Instead of single workstations, typically multiple supercomputers or clusters of workstations are connected via wide-area links, forming systems with a two-level communication hierarchy (e.g., the DAS system, see Section 1.9.1). Three-level hierarchies are created when clusters of multi-processor machines are connected via wide-area networks (e.g., the DAS-2 system, see Section 1.9.2). When running parallel applications on such multi-cluster systems, efficient execution can only be achieved when the hierarchical structure is carefully taken into account [132].

In this thesis, we will first investigate the usefulness of Remote Method Invocation (RMI) for distributed supercomputing. Remote procedure calls are proposed as a building block for grid applications in [101, 102, 150]. RMI is an object oriented form of RPC, so it should in theory also be suitable for grid programming. The Java-centric approach achieves a high degree of transparency and hides many details of the underlying system (e.g., different communication substrates) from the programmer. For many high-performance applications, however, the slow wide-area links or the huge difference in communication speeds between the local and wide-area networks is a problem. In the DAS system, for example, a Java RMI over Myrinet (a Gbit/s local area network) takes about 37 μsec, while an RMI over the ATM wide-area network costs several milliseconds. Our solution is to let the Java RMI system expose the structure of the wide-area system to the application, so applications can be optimized to reduce communication over the wide-area links. When using RMI, the application programmer must implement wide-area optimizations.

However, the ultimate goal of our work is to create a programming environment in which parallel applications for hierarchical systems are easy to implement. The application programmer should not have to implement different wide-area optimizations for each application. One possible solution of this problem is investigated in detail in this thesis. We chose one specific class of problems, divide-and-conquer algorithms, and implemented an efficient compiler and runtime system that apply wide-area optimizations automatically. Divide-and-conquer algorithms work by recursively dividing a problem into smaller subproblems. This recursive subdivision goes on until the remaining sub-problem becomes trivial to solve. After solving subproblems, their results are recursively recombined again until the final solution is assembled. Computations that use the divide-and-conquer model include geometry procedures, sorting methods, search algorithms, data classification codes, n-body simulations and data-parallel numerical programs [174].

Divide-and-conquer applications may be parallelized by letting different processors solve different subproblems. These subproblems are often called *jobs* in this context. Generated jobs are transferred between processors to balance the load in the computation.

The divide-and-conquer model lends itself well for hierarchically-structured systems because tasks are created by recursive subdivision. This leads to a hierarchically-structured task graph which can be executed with excellent communication locality, especially on hierarchical platforms. Of course, there are many kinds of applications that do not lend themselves well to a divide-and-conquer algorithm. However, we believe the class of divide-and-conquer algorithms to be sufficiently large to justify its deployment for hierarchical wide-area systems.

1.1.3   Software Solutions

In this thesis, we describe two separate high-performance Java-based infrastructures for parallel computing on geographically distributed resources. The first is a native system, called Manta, which we use for experimentation and detailed performance analysis. Because Manta is a native system, it is portable only in the traditional sense. The compiler, runtime system and application have to be recompiled for each architecture. Moreover, a separate compiler backend must be provided for each instruction set. The second system, called Ibis, is written in pure Java, and thus makes full use of Java's "write once, run everywhere" feature. The only requirement for using Ibis is a Java Virtual Machine (JVM) , which greatly facilitates deployment on grid systems. We implemented an efficient RMI mechanism as well as a divide-and-conquer extension on both systems.

Although Manta is not a complete grid computing environment yet (e.g., it currently does not provide security or fault tolerance), it is interesting for several reasons. The system focuses on optimizations to achieve high performance with Java. It uses a native compiler and an efficient, lightweight, user-level RMI implementation that achieves a performance close to that of C-based RPC protocols. Besides divide-and-conquer, Manta supports replicated objects (RepMI) and collective communication with the Group Method Invocations (GMI) mechanism.

The use of a native Java compiler as a research vehicle allows us to experiment with language extensions, communication protocols, and runtime system optimizations. A native system also allows detailed performance analysis, while JVMs are essentially "a black box", making a detailed analysis difficult. We chose to build a native compiler instead of a just-in-time (JIT) compiler because debugging and profiling is easier with statically generated code.

To solve the grid heterogeneity problem, we present Ibis, a new grid programming environment that combines Java's "run everywhere" portability both with flexible treatment of dynamically available networks and processor pools, and with highly efficient, object-based communication. Ibis can transfer Java objects very efficiently by combining streaming object serialization with a zero-copy protocol. Ibis essentially implements the optimizations that were found to be useful during experiments with Manta, but now in pure Java, thus making it ideally suited for high-performance computing in grids.

We study the use of divide-and-conquer parallelism as a paradigm for programming distributed supercomputing applications. We introduce *Satin*, which is specifically designed for running divide-and-conquer applications on multi-cluster, distributed-memory machines, connected by a hierarchical network. Satin is integrated into both Manta and Ibis. Satin's programming model has been inspired by Cilk [22] (hence its name). Satin

provides two simple primitives for divide-and-conquer programming: one to spawn work and one for synchronization, without altering the Java language. A special compiler is used to generate parallel code for Satin programs; without this compiler, Satin programs run sequentially. Satin's compiler and runtime system cooperate to implement these primitives efficiently on a hierarchical wide-area system, without requiring any help or optimizations from the programmer. Satin on top of Ibis also features well-defined exception semantics, and has support for retracting speculatively generated work. We investigate wide-area load balancing algorithms for divide-and-conquer applications which are essential for efficient execution on hierarchical systems.

We investigate two approaches for distributed supercomputing in Java. The first approach uses the standard Java programming model, in particular RMI. The second approach uses divide-and-conquer. In the remainder of this chapter, we briefly describe the systems we have built to evaluate these approaches (more detail follows in later chapters). We describe both the native Manta system which we used for experimentation, and Ibis, which solves the portability problems that come with native compilation, and is a more complete grid computing environment. Section 1.2 describes the Manta RMI system, which we use to investigate the first approach. In Section 1.3, we give a brief description of the low-level communication software that is used in this thesis. Section 1.4 describes the implementation of the Manta RMI system on the (wide-area) DAS system. Section 1.5 describes the Satin divide-and-conquer system, which we use to investigate the second approach. An overview of Ibis is given in Section 1.6. Manta and Ibis are large projects: several people have worked on them. Therefore, a short list of who did what for this thesis is presented in Section 1.7. The contributions of this thesis are given in Section 1.8. We describe the hardware that was used to obtain the experimental results in Section 1.9. Finally, we give an overview of the rest of this thesis in Section 1.10.

1.2   The Manta RMI system

Manta is a Java system designed for high-performance parallel computing. Manta uses a native compiler and an optimized RMI protocol. The compiler converts Java source code to binary executables, and also generates the serialization and deserialization routines, a technique which greatly reduces the runtime overhead of RMIs. Manta nodes thus contain the executable code for the application and (de)serialization routines. The nodes communicate with each other using Manta's own lightweight RMI protocol.

The most difficult problem addressed by the Manta system is to allow inter operability with other JVMs. One problem is that Manta has its own, lightweight RMI protocol that is incompatible with Sun's protocol. We solve this problem by letting a Manta node also communicate through a Sun-compliant protocol. Two Manta nodes thus communicate using our fast protocol, while Manta-to-JVM RMIs use the standard RMI protocol.

Another problem concerning inter operability is that Manta uses a native compiler instead of a bytecode interpreter (or JIT). Since Java RMIs are polymorphic [167], Manta nodes must be able to send and receive bytecodes to inter operate with JVMs. For example, if a remote method expects a parameter of a certain class C, the invoker may send it an object of a subclass of C. This subclass may not yet be available at the receiving Manta

node, so its bytecode may have to be retrieved and integrated into the computation. With Manta, however, the computation is a binary program, not a JVM. In the reverse situation, if Manta does a remote invocation to a node running a JVM, it must be able to send the bytecodes for subclasses that the receiving JVM does not yet have. Manta solves this problem as follows. If a remote JVM node sends bytecode to a Manta node, the bytecode is compiled dynamically to object code and this object code is linked into the running application using the operating system's dynamic linking interface. Also, Manta generates bytecodes for the classes it compiles (in addition to executable code). These bytecodes are stored at a web server, where remote JVM nodes can retrieve them.

Manta also supports JavaParty[131] style remote objects. These objects are marked with a special *remote* keyword, and can be created on remote machines. This model is slightly different from the standard RMI model. In Manta, the *remote* keyword is syntactic sugar to make the use of remote objects somewhat more convenient; the communication implementation for RMI and JavaParty style remote objects, however, is the same. For simplicity, we will only use the standard (well known) RMI model in this thesis.

1.3   Communication software

RMI implementations are typically built on top of TCP/IP (Transmission Control Protocol/Internet Protocol), which was not designed for parallel processing. Therefore, we use different communication software for Manta. To obtain a modular and portable system, Manta is implemented on top of a separate communication library, called Panda [12, 147]. Panda provides communication and multithreading primitives and is designed to be used for implementing runtime systems of various parallel languages. Panda's communication primitives include point-to-point message passing, RPC, and multicast. The primitives are independent of the operating system or network, which eases porting of runtime systems (for programming languages) implemented on top of Panda. The implementation of Panda is structured in such a way that it can exploit useful functionality provided by the underlying system (e.g., reliable message passing or multicast), which makes communication efficient [12]. If these features are not present in the lower layers, Panda implements them in software. Panda uses a scatter/gather interface to minimize the number of memory copies, resulting in high throughput.

The Panda message passing interface is based on an *upcall* model: conceptually a new thread of control is created when a message arrives, which will execute a handler for the message. The interface has been designed to avoid context switches in simple, but frequent, cases. Unlike active message handlers [49], upcall handlers in Panda are allowed to block to enter a critical section, but a handler is not allowed to wait for another message to arrive. This restriction allows the implementation to handle all messages using a single (the currently running) thread, so handlers that execute without blocking do not need any context switches.

Panda has been implemented for a variety of machines, operating systems, and networks. For communication over Myrinet, Panda internally uses the LFC communication system [18] on DAS, and GM[1] (Myricom's standard software for Myrinet) on DAS-2.

---

[1]See http://www.myri.com.

LFC is a reliable, highly efficient communication substrate for Myrinet, similar to active messages. LFC was designed to provide the right functionality for parallel programming systems, and it has been used to implement a variety of higher-level systems, including Orca [12], data-parallel Orca [72], MPI [89], and CRL [81]. LFC itself is implemented partly by embedded software that runs on the Myrinet Network Interface processor and partly by a library that runs on the host processor. To avoid the overhead of operating system calls, the Myrinet Network Interface is mapped into user space, so LFC and Panda run entirely in user space. The current LFC implementation does not offer protection against malicious programs, so the Myrinet network can be used by a single process only. (This problem can be solved with other Myrinet protocols that do offer protection [18].) On Fast Ethernet, Panda is implemented on top of UDP (User Datagram Protocol), using a 2-way sliding window protocol to obtain reliable communication and flow control. The Ethernet network interface is managed by the kernel (in a protected way), but the Panda RPC protocol runs in user space.

GM is Myricom's standard software for Myrinet. In contrast to LFC, it does not specifically target runtime systems for programming languages. GM consists of a kernel driver, some software that runs on the network interface card and some software that runs on the host. Although GM provides efficient user-level communication primitives, the interface lacks features that are important for parallel programming systems, such as multicast and support for interrupts or upcalls.

1.4   Manta on the (wide-area) DAS system

When implementing Java communication on a wide-area system like DAS, the most important problem is how to deal with the different communication networks that exist within and between clusters. We assume that wide-area parallel systems are hierarchically structured and consist of multiple parallel machines (clusters or Massively Parallel Processors (MPPs)) connected by wide-area networks. To fully exploit the high performance of the Local Area Network (LAN) within a cluster (or MPP interconnects), it is important that the communication protocols used for intra-cluster communication are as efficient as possible. Inter-cluster communication over the Wide Area Network (WAN) necessarily is slower.

Most Java RMI implementations are built on top of TCP/IP. Using a standard communication protocol eases the implementation of RMI, but also has a major performance penalty. TCP/IP was not designed for parallel processing, and therefore has a high overhead on fast LANs such as Myrinet. For the Manta system, we therefore use different protocols for intra-cluster and inter-cluster communication. The implementation of Manta and Panda on the wide-area DAS system is shown in Figure 1.1. For intra-cluster communication over Myrinet, Panda internally uses LFC on DAS, and GM on DAS-2.

For inter-cluster communication over the wide-area network, Panda uses one dedicated *gateway* machine per cluster. The gateways also implement the Panda primitives, but support communication over both Myrinet and TCP/IP. A gateway communicates with the machines in its local cluster, using LFC or GM over Myrinet. In addition, it communicates with gateways of other clusters, using TCP/IP. The gateway machines thus
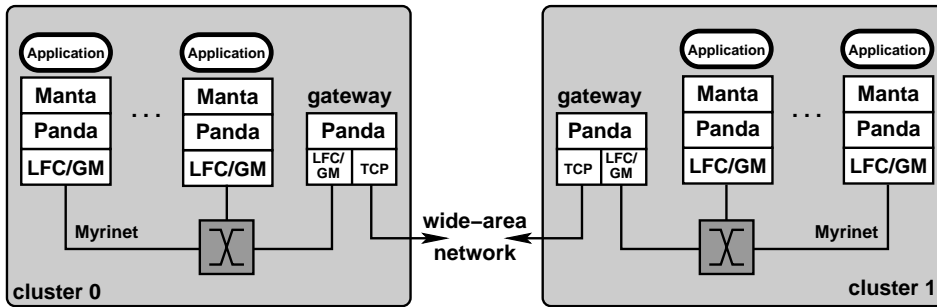
Figure 1.1: Wide-area communication based on Panda.

forward traffic to and from remote clusters. In this way, the existence of multiple clusters is transparent to the Manta runtime system. Manta's protocols simply invoke Panda's communication primitives, which internally call LFC, GM and/or TCP/IP primitives.

The resulting Java system thus is highly transparent, both for the programmer and the runtime system implementor. The system hides several complicated issues from the programmer. For example, it uses a combination of active messages and TCP/IP, but the application programmer sees only a single communication primitive (e.g., RMI or Satin). Likewise, Java hides any differences in processor types from the programmer.

As stated before, parallel applications often have to be aware of the structure of the wide-area system, so they can minimize communication over the wide-area links. Manta RMI programs therefore can find out how many clusters there are and to which cluster a given machine belongs. In Chapter 4, we will give several examples of how this information can be used to optimize programs. Satin programs need not be aware of the cluster topology, as the Satin runtime system is cluster aware, and tries to balance the load in the system.

1.5   The Satin Divide-and-Conquer System

While the first part of this thesis investigates parallel programming on grids using RMI, the second approach to distributed supercomputing studied in this thesis is to use the divide-and-conquer programming model. We have implemented this model by extending the Manta native Java compiler and runtime system. The resulting system is called Satin. To achieve a "run everywhere" implementation, Satin was also implemented on top if Ibis.

We implemented several optimizations to achieve good performance. First, it is imperative to have efficient local execution (i.e., on a single machine). When the overhead of Satin's primitives is already high on a single machine, there is little use in running the application in parallel. Satin achieves good performance in the local (one-processor) case because the parameter semantics of the language are designed to allow an efficient implementation. This way, the Satin runtime system can reduce the overhead of local jobs

Hierarchy level	Performance issues
Local	Spawning and synchronization overhead, locking the work queue.
Cluster	Efficient communication and serialization.
Wide-area	Load balancing.

Table 1.2: Performance issues when implementing a divide-and-conquer system.

using on-demand serialization, which avoids copying and serialization of parameters for jobs that are not transferred to a remote machine. Furthermore, the local case is optimized by using efficient locking mechanisms for the job queue, and by letting the Manta and Ibis compilers generate code to efficiently implement the Satin primitives.

Second, performance and scalability on a single cluster is important. Satin achieves good performance in this case by exploiting the efficient user-level communication protocols that are also used for the Manta RMI implementation. Manta's efficient, compiler-generated serialization code is reused in Satin to marshall parameters of spawned method invocations that are executed on a remote machine.

Finally, the performance on multiple clusters should be better than the performance of a single cluster, otherwise it would be useless to run the applications on a multi-cluster environment such as the grid. Load balancing algorithms that are used in single cluster systems fail to achieve good performance on wide-area systems. We found that load-balancing algorithms that are proposed in the literature for wide-area systems also perform suboptimally. Therefore, Satin uses a novel load-balancing algorithm, called *Cluster-aware Random Stealing* (CRS), which performs well on clustered wide-area systems. The performance issues that occur when designing a divide-and-conquer implementation for hierarchical systems are summarized in Table 1.2.

1.6 Ibis

In computational grids, performance-hungry applications need to simultaneously tap the computational power of multiple, dynamically available sites. The crux of designing grid programming environments stems exactly from the dynamic availability of compute cycles: grid programming environments (a) need to be *portable* to run on as many sites as possible, (b) they need to be *flexible* to cope with different network protocols and dynamically changing groups of compute nodes, while (c) they need to provide *efficient* (local) communication that enables high-performance computing in the first place.

Existing programming environments are either portable (Java), or they are flexible (Jini, Java RMI), or they are highly efficient (MPI). No system combines all three properties that are necessary for grid computing. In this thesis, we present Ibis, a new programming environment that combines Java's "run everywhere" portability both with flexible treatment of dynamically available networks and processor pools, and with highly efficient, object-based communication. Because Ibis is Java based, it has the advantages that come with Java, such as portability, support for heterogeneity and security. Ibis has been designed to combine highly efficient communication with support for both heterogeneous

software component	implementor(s)	see also
the native Manta system:		
the native compiler	Ronald Veldema and Ceriel Jacobs	[165]
Manta serialization (compiler + RTS)	*Rob van Nieuwpoort*	Chapter 3
Manta RMI (compiler + RTS)	*Rob van Nieuwpoort*	Chapter 3
garbage collector	Jason Maassen	-
distributed garbage collector for RMI	Jason Maassen	Chapter 3
Sun compatibility code:		
Sun compatible serialization	Jason Maassen	Chapter 3
bytecode loader	Rutger Hofman	Chapter 3
JNI	Rutger Hofman	-
Sun compatible RMI	Ceriel Jacobs	Chapter 3
Satin:		
Satin compiler	*Rob van Nieuwpoort*	Chapter 5
Satin runtime system	*Rob van Nieuwpoort*	Chapter 5
Satin load balancing	*Rob van Nieuwpoort*	Chapter 6
Manta/Ibis:		
Ibis design	*Rob van Nieuwpoort* and Jason Maassen	Chapter 7
Ibis implementation on top of TCP	*Rob van Nieuwpoort* and Jason Maassen	Chapter 7
Ibis implementation on top of Panda	Rutger Hofman	Chapter 7
Satin to bytecode compiler for Ibis	*Rob van Nieuwpoort*	Chapter 8
Satin runtime system on top of Ibis	*Rob van Nieuwpoort*	Chapter 8
Manta related projects:		
RepMI (both on Manta and Ibis)	Jason Maassen	[109]
GMI (both on Manta and Ibis)	Jason Maassen	[109]
Jackal DSM	Ronald Veldema	[165]

Table 1.3: Who did what for this thesis.

networks and malleability (adding and removing resources during the computation). Ibis can be configured dynamically at run time, allowing to combine standard techniques that work "everywhere" (e.g., TCP) with highly-optimized solutions that are tailored for special cases, like a local Myrinet interconnect. Ibis can transfer Java objects efficiently by combining streaming object serialization with a zero-copy protocol. Using RMI as a simple test case, we show (in Section 7.3) that Ibis outperforms existing RMI implementations, achieving up to 9 times higher throughputs with trees of objects.

1.7   Who Did What for this Thesis

Manta and Ibis are large projects. Jason Maassen, Ronald Veldema and the author closely cooperated on the Manta system, and both Jason Maassen and the author worked together on Ibis. It is therefore sometimes hard to indicate where the boundaries of our respective contributions lie. Two programmers in our group, Ceriel Jacobs and Rutger Hofman, also worked on the Manta system. This section explains in some detail who did what on the Manta system. The section is summarized in Table 1.3.

The author designed and implemented the Manta object serialization mechanism, which is used to marshall parameters to Manta RMIs. The author also designed the Manta

RMI system and implemented the initial version. Rutger Hofman further optimized the compiler-generated serialization and marshalling code to make them even more efficient. Both the object serialization and the RMI system are partly implemented in the native compiler and partly in the runtime system (RTS). Also, both components interface with Panda, the communication substrate we use. The Manta RMI system also interfaces with the garbage collector. There was extensive cooperation with the people who implemented those parts. The author also implemented all parts of the Satin system, both in the runtime system and in the Manta compiler.

Jason Maassen implemented Manta's garbage collector and distributed garbage collector for Manta RMI. He also implemented the serialization code that is compatible with the Sun standard. This serialization is used for Sun compatible RMI. RepMI, a Java extension for replicated objects was also designed and implemented by Jason. The replicated objects can be used in combination with Satin and are therefore relevant for this thesis. Jason also implemented a system with collective communication operations, called GMI. GMI is not used or discussed in this thesis.

Jason and the author worked together where the code that we wrote had to inter operate, for example Manta RMI and the garbage collector. We also worked together to integrate Satin and RepMI. Furthermore, we designed Ibis together, with helpful suggestions from Rutger Hofman. We wrote the TCP/IP implementation of Ibis together. Rutger wrote the Panda implementation of Ibis. When Ibis was finished, the author implemented Satin on top of it, and Jason likewise implemented the replicated objects (RepMI) and GMI.

Ronald Veldema and Ceriel Jacobs implemented most of the Manta native compiler. Because the Manta native serialization, RMI, and also parts of Satin are partly compiler generated, the author worked together with Ronald during the implementation of these systems. Rutger Hofman implemented the bytecode loader which is used to implement heterogeneous Sun RMI calls. He also implemented the Java Native Interface (JNI) for Manta. Furthermore, both Ceriel Jacobs and Rutger Hofman implemented improvements in all areas of the compiler and runtime system.

1.8   Contributions

This thesis presents a number of new ideas. We can summarize them as follows.

1. Using the Manta system, we show that object-based communication in Java, and in particular RMI, can be made highly efficient.

2. Using Ibis, we demonstrate that this is even possible while maintaining the portability and heterogeneity features (i.e., "write once run everywhere") of Java.

3. We demonstrate that it is possible to write efficient, fine grained distributed super-computing applications with Java RMI, but that the programmer has to implement application-specific wide-area optimizations.

4. With Satin, we integrate divide-and-conquer primitives into Java, without changing the language.

5. We show that, using serialization on demand and user-level (zero-copy) communication, an efficient divide-and-conquer system that runs on distributed memory machines and the grid can be implemented in Java.

6. We demonstrate that divide-and-conquer applications can be efficiently executed on hierarchical systems (e.g., the grid), without any wide-area optimizations by the application programmer, using novel wide-area-aware load-balancing algorithms.

7. We present a mechanism that is integrated into Java's exception handling model, and that allows divide-and-conquer applications to abort speculatively spawned work on distributed memory machines.

8. We validate our claim that the Java-centric approach greatly simplifies the deployment of efficient parallel applications on the grid. We do this by running a parallel application on a real grid testbed, using machines scattered over the whole of Europe.

1.9   Experimental Environment

We used two different hardware platforms, called DAS (Distributed ASCI[2] Supercomputer) and its successor DAS-2, for the experimental results presented in this thesis. We believe that high-performance grid computing applications will typically run on collections of parallel machines (clusters or SMPs), rather than on workstations at random geographic locations. Hence, grid computing systems that are used for parallel processing will be *hierarchically* structured. The DAS experimentation systems, built by the ASCI research school, reflect this basic assumption. Both DAS systems consist of several clusters located at different universities in The Netherlands. The DAS has a two level hierarchy (there are communication links within the clusters and between the clusters), while DAS-2 has a three level hierarchy, because the clusters contain dual processor machines. We will discuss the two systems in more detail below.

1.9.1   DAS

The structure of the DAS is shown in Figure 1.2. It consists of four clusters. The nodes within the same cluster are connected by 1.2 Gbit/sec Myrinet [25]. The clusters are connected by dedicated 6 Mbit/s wide-area ATM networks, which were later replaced by regular Internet links with more bandwidth.

The processors in each node is a 200 MHz Pentium Pro, with at least 64 MBytes memory. The cluster at the Vrije Universiteit has 68 processors and the other clusters have 24 nodes each. The machines run RedHat Linux 6.2 (kernel version 2.2.14-12). The Myrinet (a popular multi-Gigabit LAN) network is a 2D torus and the wide-area network is fully connected.

---

[2]The ASCI (Advanced School for Computing and Imaging) research school is unrelated to, and came into existence before, the Accelerated Strategic Computing Initiative.
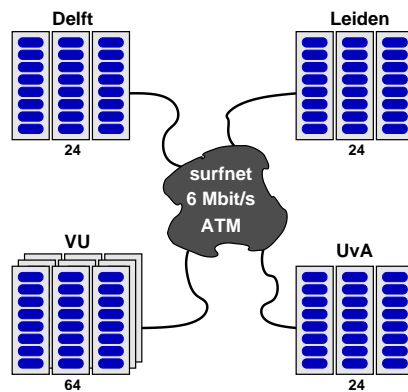
Figure 1.2: The wide-area DAS system.

1.9.2   DAS-2

DAS-2 is a wide-area distributed computer of 200 Dual Pentium-III nodes. The machine is built out of five clusters of workstations, which are interconnected by SurfNet, the Dutch academic Internet backbone for wide-area communication, whereas Myrinet is used for local communication. The overall structure of the system is similar as is shown in Figure 1.2, with an additional cluster located at the University of Utrecht.

The cluster at the Vrije Universiteit contains 72 nodes; the other four clusters have 32 nodes each (200 nodes with 400 CPUs in total). The system was built by IBM. The operating system we used is RedHat Linux 7.2 (kernel version 2.4.9-13.smp). Each node contains two 1.00-GHz Pentium-IIIs with at least 1 GByte RAM, a local IDE disk of least 20 GByte and a Myrinet interface card as well as a Fast Ethernet interface (on-board). The nodes within a local cluster are connected by a Myrinet-2000 network, which is used as high-speed interconnect, mapped into user-space. Myrinet is used by the parallel applications, while Fast Ethernet is used as operating system network (e.g., file transport).

1.10   Outline of this Thesis

Chapter 2 provides background material. Some general related work is discussed, and we give explanations of topics that are needed to understand this thesis. First, we discuss Java and its features, such as portability, threads, synchronization and RMI. Next, we give an overview of the divide-and-conquer programming model. Finally, we give a brief description of the application programs used for performance analysis in this thesis.

Chapter 3 describes the Manta serialization and RMI implementations in detail. The chapter is based on the work described in [113] and [112]. We focus on the differences between RPC and RMI (e.g., polymorphism), and the problems that are introduced by them. We describe both a Sun compatible RMI implementation and a highly optimized, but incompatible implementation that is used between Manta nodes. For inter operabil-

ity, Manta can dynamically switch between the two implementations. We analyze the performance of both implementations in detail.

In Chapter 4, which is based on [126] and [127], we investigate the usefulness of RMI for distributed supercomputing. We present a case study of four RMI applications that we optimized for hierarchical grids. We find that application-specific optimizations that exploit the hierarchical structure of the system are required.

In Chapter 5, we introduce the divide-and-conquer system Satin, implemented in the Manta system. This chapter is based on work that was described earlier in [124]. We focus on performance on a single machine and on a cluster of workstations. The performance of Satin on clusters is important, because we believe that the grid will be hierarchical in nature, and will consist of connected clusters of workstations.

We show how Satin is optimized for hierarchical grids in Chapter 6. Parts of this chapter have been published in [125] and [88]. The most important problem that was solved is balancing the load of the computation in the presence of fast local communication and slow wide-area links. We investigate five load balancing algorithms in detail, using different applications and various wide-area latencies and bandwidths. We demonstrate that special, wide-area-aware load-balancing algorithms are required for good performance.

In Chapter 7, we present Ibis, a new grid computing programming environment in pure Java. This chapter is based on work described in [128]. Ibis implements the optimizations that were found useful in our experiments with the Manta system. Ibis exploits Java's "write once, run everywhere" features to achieve portability, but still allows the use of efficient (user-level) communication substrates, using runtime configuration and dynamic class loading.

Chapter 8 describes the Satin implementation on top of Ibis. The Satin compiler and runtime system are also completely written in Java to maintain the portability features that Java and Ibis provide. We present a case study that investigates Satin in a real grid environment. Moreover, we investigate a new mechanism for aborting speculative computations that have become superfluous. The new mechanism is cleanly integrated with Java's exception handling mechanism, and allows a larger class of applications to be expressed in Satin.

Finally, in Chapter 9, we draw our conclusions. We determine to what extend we were able to fulfill our research goals. Furthermore, we present areas where future research is useful.

Chapter 2

Background and Related Work

```
Wise men say, and not without reason, that whosoever wished to
foresee the future might consult the past.
```

```
- Niccolo Machiavelli
```

In this chapter, we explain the background that is needed to read this thesis. We explain what Java is, what useful features it has (e.g., portability, threads, and RMI), and why we use it for grid computing. We also give an overview of grid computing, divide-and-conquer programming and load balancing. Furthermore, we discuss work in these areas that does not directly relate to our work, but which is interesting background material. Closely related work is discussed separately in subsequent chapters. Finally, we give a short description of the application programs that are used for performance measurements in this thesis. If the reader is familiar with the subjects presented here, it is possible to skip this chapter, because it is intended to provide some context for the work in this thesis, and does not contain any new research.

2.1 Grid Computing

Foster and Kesselman give a definition of the grid in [57]:

> The *grid* is an emerging infrastructure that will fundamentally change the way we think about –and use– computing. The grid will connect multiple regional and national *computational grids* to create a universal source of computing power. The word "grid" is chosen by analogy with the electric power grid, which provides pervasive access to power and, like the computer and a small number of other advances, has had a dramatic impact on human capabilities and society.

The goals of the grid are to provide pervasive, dependable, consistent and inexpensive access to advanced computational capabilities, databases, scientific instruments, and people. Foster and Kesselman provide an excellent starting point for research on the grid in [57].

The Globus project [56] is a research and development project focused on enabling the application of grid concepts to scientific and engineering computing [1]. A part of the project is the development of the Globus Toolkit, a set of services and software libraries to support grids and grid applications. The Globus toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection, and portability, and has become the de-facto standard in grids.

Legion [70] is an object-based, meta-systems software project at the University of Virginia[2]. The system addresses key issues like scalability, programming ease, fault tolerance, security, and site autonomy. Legion is designed to support large degrees of parallelism in application code and to manage the complexities of the physical system for the user.

Condor [161] is a project that aims to develop, implement, deploy, and evaluate mechanisms and policies that support High Throughput Computing (HTC) on large collections of distributively owned computing resources[3]. Condor provides a job-queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. It is also possible to combine several Condor pools into a single system [51], thus creating a "flock" of Condors, possibly connected by a wide-area network.

Distributed supercomputing is a subclass of grid computing, and deals with parallel computing on geographically distributed resources. This thesis discusses a Java-centric approach to writing wide-area parallel (distributed supercomputing) applications. Most other grid computing systems (e.g., Globus [56] and Legion [70]) support a variety of languages. The SuperWeb [1], the Gateway system [73], Javelin 3 [120], the skeleton programming model introduced by Alt et al. [2], and Bayanihan [149] are examples of global computing infrastructures that support Java. A language-centric approach makes it easier to deal with heterogeneous systems, since the data types that are transferred over the networks are limited to the ones supported by the language (thus obviating the need for a separate interface definition language) [171].

EveryWhere [172] is a toolkit for building adaptive computational grid programs[4]. It contains a set of software tools designed to allow an application to leverage resources opportunistically, based on the services they support. By using the EveryWare toolkit, a single application can simultaneously combine the useful features offered by Globus, Legion, Condor, Java applets, and other systems.

2.2   Java

Java [68] is an object-oriented programming language that is syntactically similar to C and C++. Java has several features that make it a convenient language to program, some of which we will describe below. Furthermore, we will explain why Java is especially useful for grid computing.

---

[1]See *http://www.globus.org/* for more information on Globus.
[2]See *http://legion.virginia.edu/* for more information on Legion.
[3]See also *http://www.cs.wisc.edu/condor/*.
[4]See also *http://nws.npaci.edu/EveryWare/*.

2.2.1   Ease of Use

Since Java is an object-oriented programming language, it supports well-known features like classes and interfaces, inheritance, dynamic binding, overloading, and polymorphism. Memory management is straightforward in Java because it uses garbage collection. Furthermore, runtime cast checks (to ensure that classes are not converted to an incompatible type), the absence of pointer arithmetics, and array-bound checking make it impossible to corrupt the heap, thus effectively ruling out a large class of programming errors that are difficult to find.

Java uses exceptions to force the programmer to handle runtime errors. Exceptions are a part of the class hierarchy, and are thus typed, and can be extended with user defined exceptions. Methods must specify which exceptions they potentially throw. These exceptions must be handled by the caller. There is a class of exceptions that do not have to be caught (all subclasses of *java.lang.RuntimeException*), such as an array bound check that failed. Exceptions contain a stack trace, which makes it easier to find the problem.

Java is a flexible language, because of the class loading mechanism. This mechanism can be used to dynamically load new code into the running application. This feature allows polymorphism in RMIs, as will be discussed in Section 2.3.3.

Java comes with an extensive set of class libraries [35–37] that make it easy to build complex applications. Data structures like hash tables, lists, etc. are predefined. The class libraries are well documented with Java's source code documentation system, called javadoc. All aforementioned features make Java easy to learn and use. This is important, as we want to use Java to enable grid computing. Many programmers who write grid applications are not computer scientists by origin.

2.2.2   Portability

Instead of compiling the Java source to executable code for a given platform, Java is compiled to a machine-independent format, called *bytecode* [105]. This bytecode is interpreted or compiled just-in-time (JIT, or compilation at run time) by a Java virtual machine [105] (JVM). Compiled Java programs are shipped in the platform independent bytecode format, and can be executed on any architecture, provided that a JVM is present for the target platform. Although new features were added to Java since its introduction (e.g., inner classes), the bytecode format has never changed.

Java source code can also be compiled directly to executable code for a platform (as our Manta system does), but this way, the advantages of Java bytecode (e.g., portability) are lost. When the work on this thesis started, the performance of JVMs was poor. For many applications, there was an order of magnitude difference between the performance of Java code and compiled C code. Because we wanted to use Java for high-performance computing, we needed efficient execution of Java code. The Manta system we created provides an efficient compiler and runtime system, together providing an efficient Java platform. However, during the time that the research for this thesis was done, the performance of JVMs improved dramatically. Just-in-time compilers now provide excellent performing Java platforms, and the performance of just in time compiled Java code comes close that of statically compiled C code [29, 141]. Some researchers believe that the per-

formance of Java JITs will even overtake statically compiled C code, as a JIT can use runtime information to optimize the performance of the running program [134]. For instance, runtime constants can be detected and propagated, leading to more efficient code.

The fact that Java uses bytecode to distribute executables is not sufficient to ensure better portability than traditional languages provide. One could argue for instance, that ANSI C is also portable. However, some fundamental language characteristics such as the primitive types (*int*, *long*, *double*, etc.) are not well defined. An integer value in a C program is defined as "at least 16 bits", therefore, it may be 32 bits, but it may also be 64 or 31. The same holds for *long* and *double* values. Java defines the sizes of all primitive types. Other features that are not well defined in C are, for instance, data alignment, byte ordering and the rounding and precision of floating point numbers. Although it is possible to write portable C programs, this requires enormous discipline of the programmer, because many language constructs are allowed, but undefined.

Furthermore, many non-trivial programs need several libraries to work. These libraries are often non-standard, and header files that describe the functionality of the libraries can reside at different locations for different platforms. Moreover, there are often several versions of the same library in use at the same time over different sites. In practice, many C programs are not portable because of library issues. These issues can be partly fixed by using preprocessor directives (#if, #ifdef, etc.), but different directives are then needed for each platform. For instance, during this thesis, we migrated the DAS from the BSDI operating system to Linux, and found that it took a large effort to port the Manta compiler and runtime system, which are written in C++ and C, even though the underlying hardware remained the same. Furthermore, we found that special care had to be taken to ensure that Manta compiles on different distributions of the same Linux version.

Moreover, we ran into several bugs in the C++ compilers we tried, especially when more advanced features are used (e.g., C++ templates) or when compiler optimizations are enabled (e.g., function inlining). Furthermore, due to the enormous complexity of the C++ language, at the time of writing there is exactly one compiler frontend [48] that implements the complete C++ standard (ISO/IEC 14882:1998). All others (e.g., GNU G++, Microsoft VC++) only partially implement the standard, leading to poor portability of C++ code. We can conclude that C and C++ programs are often not very portable in practice due to library and compiler issues.

Java solves the library problem by defining a large set of standard classes in a class library. Packages are provided for doing file I/O, networking (UDP and TCP), Serialization (ObjectStreams), graphical interfaces (AWT and Swing), and much more. These packages must be present in all Java systems. Different versions of the class libraries exist, but they are backward compatible. By compiling source code to bytecode, and by strictly defining the language and class libraries, Java achieves true portability between platforms. Furthermore, Java is a much cleaner language than C++. It is therefore much easier to write a Java compiler. Java bytecode is also quite simple and concise, simplifying the development of JVMs. Due to Java's simplicity, excellent compilers, runtime systems, debuggers and other tools have been developed in a short time frame.

```java
1  class MyThread1 extends java.lang.Thread {
2      int n;
3
4      MyThread1(int n) {
5          this.n = n;
6      }
7
8      // This method is executed when the thread is started.
9      public void run() {
10         System.out.println("n = " + n);
11         // Body of the thread...
12     }
13
14     public static void main() {
15         MyThread1 t = new MyThread1(42); // Create MyThread1 object.
16         t.start(); // Start the thread.
17     }
18 }
```

Figure 2.1: A Java threading example, extending java.lang.Thread.

2.2.3 Security

An important issue for grid computing is security. Many sites are reluctant to run arbitrary user code on their machines, due to security issues, while this is of critical importance to the grid. Java addresses these security problems in several ways. First, when Java code is loaded into the virtual machine, the bytecode is verified. This way, the JVM ensures that no illegal operations are performed at the bytecode level. Second, Java applets and applications can be sandboxed. This way, all code that, for instance, tries to access files or send network messages, is checked before access to the requested resources is allowed. The site administrator can create policies that describe in detail which resources may be accessed.

2.3 Parallel Programming in Java

Java provides two mechanisms that can be used for parallel programming: multithreading for shared memory architectures, and RMI for distributed memory architectures. This thesis focuses on RMI, because efficient shared memory is not feasible when systems are so widely distributed as they are on the grid. Still, we discuss Java's threading mechanism here, as it is often used in combination with RMI. Because RMI is synchronous, threads must be used to simulate asynchronous communication. Furthermore, we discuss Java threads, because this allows us to (in Chapter 5) show the differences with the Satin threading model.

```
1  class MyThread2 implements java.lang.Runnable {
2      int n;
3
4      MyThread2(int n) {
5          this.n = n;
6      }
7
8      // This method is executed when the thread is started.
9      public void run() {
10          System.out.println("n = " + n);
11          // Body of the thread...
12      }
13
14      public static void main() {
15          MyThread2 t = new MyThread2(42); // Create MyThread2 object.
16          new Thread(t).start(); // Start a new thread.
17      }
18  }
```

Figure 2.2: A Java threading example, implementing java.lang.Runnable.

2.3.1   Threads, Locks and Condition Synchronization

A useful feature of Java is that the threading mechanism is embedded in the language. This way, threads are cleanly integrated with the object-oriented programming model and more importantly, the Java memory model is designed to support efficient implementations of threads on SMPs and DSM (Distributed Shared Memory) systems.

There are two ways in Java to create new threads. The first is to create an object of a class that extends the *java.lang.Thread* class. When the *start* method of this object is called, a new thread is started that executes the user defined *run* method. An example of this is shown in Figure 2.1. The second way is to create a new *java.lang.Thread* object and pass an object that extends the *java.lang.Runnable* interface as an argument to the constructor. The way this works is shown in Figure 2.2. The latter case is useful, because Java does not offer multiple inheritance. Using the *java.lang.Runnable* interface, allows the programmer to derive the *MyThread2* class from another class (besides *java.lang.Thread*). In both cases, data can be passed to the thread via the wrapper classes *MyThread1* and *MyThread2*.

Figure 2.3 shows a more advanced threading example, which we will use to explain locks and the wait/notify construct for condition synchronization. Of course, when using threads, locks are needed to protect shared data. Java does not export locks. Instead any object can be used as a lock. Java provides the *synchronized* modifier and keyword to mark critical regions. This way the lock and unlock operations are hidden from the programmer. Thus, lock and unlock cannot be incorrectly paired, and unlock operations cannot be forgotten. The use of a synchronized keyword is shown in Figure 2.3, for instance at the lines 35–38. The synchronized modifier can also be placed in front of a method and indicates that the entire method is a critical region (see Figure 2.3, line 6). The most important difference between Java's synchronization and classic monitors is

```
1  class FibThread extends java.lang.Thread {
2      long n, result = -1;
3
4      FibThread(long n) { this.n = n; }
5
6      public synchronized long getResult() {
7          while(result == -1) {
8              try {
9                  wait();
10             } catch (InterruptedException e) {
11                 System.out.println("We were interrupted! " + e);
12             }
13         }
14         return result;
15     }
16
17     public void run() { // Called when the thread is started.
18         if (n < 2) {
19             result = n;
20         } else { // Create FibThread objects.
21             FibThread t1 = new FibThread(n-1);
22             FibThread t2 = new FibThread(n-2);
23
24             t1.start(); t2.start(); // Start the threads.
25
26             long x = t1.getResult(); // Wait for the sub-results.
27             long y = t2.getResult();
28             result = x + y;
29         }
30         synchronized(this) {
31             notifyAll(); // Notify waiting thread: result is ready.
32         }
33     }
34
35     public static void main(String[] args) {
36         FibThread t = new FibThread(10);
37         t.start();
38         long result = t.getResult();
39         System.out.println("result of fib 10 = " + result);
40     }
41 }
```

Figure 2.3: A Java threading example: Fibonacci.

that Java's locks are reentrant: a thread is allowed to aquire the same lock multiple times.

The wait/notify construct can be used for conditional thread synchronization. The *wait()* operation is invoked on an object, and causes the current thread to wait until another thread signals it by invoking either the *notify()* or the *notifyAll()* method on the object. When invoking *wait()* on an object, the current thread must own the object's lock, which is then released by the *wait()*. When the thread is notified (again, the notifier must own the lock), the wait operation recaptures the lock, and the method can continue. The use

of wait/notify is explained in Figure 2.3. The *getResult()* method on the lines 6–15 waits (line 9) until a result is produced by another thread. When the producer thread sets the result, it calls *notifyAll()* (line 22 and 37) to inform the (possibly) waiting thread. In this example program, the *Thread.join()* operation could be used as alternative for the needed synchronization. The *join()* method is used to wait for a thread to finish. The wait/notify construct is more generic: both threads continue after the synchronization.

2.3.2   Serialization

Serialization is a mechanism for converting (graphs of) Java objects to a stream of bytes. The serialization specification [159] describes how serialization works, and strictly defines the byte format. The resulting byte stream is platform independent. Therefore, serialization can be used to ship objects between machines, regardless of the architectures. One of the nice features of Java serialization is that the programmer does not have to do anything besides letting the objects that need to be serialized implement the special *java.io.Serializable* interface. This "marker" interface contains no methods, so no special serialization code has to be written by the application programmer. Instead, the JVM recognizes classes that implement *java.io.Serializable* and handles their serialization.

The serialization mechanism always makes a deep copy of the objects that are serialized. For instance, when the first node of a linked list is serialized, the serialization mechanism traverses all references contained in the first node, and serializes the objects the references point to. In this case, the reference points to the next (second) node. Thus this node is also serialized. Because the second node has a reference to the third node, that node also gets serialized, and so on, until the entire list is converted into a stream of bytes. This recursive behavior can be avoided by the programmer. If a reference in an object is declared to be *transient*, the reference will not be traversed. The serialization mechanism can also handle cycles, so in fact arbitrary data structures can be converted to a stream of bytes.

When objects are serialized, not only the object data is converted into bytes, but type and version information is also added. This way, the versions and types can be checked when the stream is deserialized. When a version or type is unknown, the deserializer can use the bytecode loader to load the correct class file for the type into the running application.

Serialization and its performance is of critical importance for parallel programming, as it is used to transfer objects over the network (e.g., parameters to remote method invocations for RMI, and parameters to spawned method invocations for Satin). Serialization, its performance, and related work are discussed in more detail in Chapter 3.

2.3.3   RMI

RMI or Remote Method Invocation [157] is an object-oriented form of the remote procedure call (RPC) [158, 167]. While RMI was originally intended for client-server applications (distributed programming), we investigate its use for parallel programming in this thesis. We think that the RMI model is suitable for parallel programming, because it integrates cleanly into Java's object-oriented programming model, and because RMI is more

```
1  public interface RemoteHello extends java.rmi.Remote {
2      public int sayHello(Object msg) throws java.rmi.RemoteException;
3  }
```

Figure 2.4: A Java interface that tags the method *sayHello* as being remote.

```
1  public class RemoteHelloImpl
2              extends java.rmi.server.UnicastRemoteObject
3              implements RemoteHello {
4
5      public RemoteHelloImpl() throws java.rmi.RemoteException {}
6
7      public int sayHello(Object msg) throws java.rmi.RemoteException {
8          System.out.println("I received a message from a client:");
9          System.out.println(msg.toString());
10         return 42;
11     }
12 }
```

Figure 2.5: An implementation of the *RemoteHello* interface.

flexible than RPC (and RPC-like systems such as CORBA), as it supports polymorphism and exceptions.

We will explain the RMI programming model with an example. As shown in Figure 2.4, the programmer can define that methods are to be called remotely (i.e., via RMI), using a special marker interface. In this example, we define a new interface, called *RemoteHello*, which contains the method *sayHello*. The interface extends the marker interface *java.rmi.Remote*, thus indicating that the methods mentioned in this interface are to be called via RMI, instead of being normal invocations. All methods that are called via RMI must have a *throws* clause that contains *java.rmi.RemoteException*. This way, Java forces the programmer to handle errors that might occur due to network link failures, etc. An advantage of using standard Java interfaces to define which methods are to be called remotely is that no separate interface description language (IDL) is needed [171]. RPC and CORBA for instance, do require a special IDL. A drawback of this is that only types that are defined in the IDL can be passed as a parameter to the remote invocation, while with RMI, *any object of any type* may be passed as a parameter. Some IDLs allow non-standard types to be passed as a parameter, but then the pack and unpack functions have to be written by hand. Another difference between RMI and traditional RPC systems is that RMI makes a deep copy of the parameters (because serialization is used), while traditional RPC mechanisms can only send shallow copies of the data.

The implementation of the remote object is shown in Figure 2.5. The remote object is called *RemoteHelloImpl* and implements the *RemoteHello* interface defined in Figure 2.4. Furthermore, the class *java.rmi.server.UnicastRemoteObject* should be extended by remote objects. Because the *RemoteHelloImpl* class implements the *RemoteHello* interface, it must implement the *sayHello* method specified there. In this example, the method just prints the *msg* parameter.

```
1  import java.rmi.*;
2  import java.rmi.registry.*;
3
4  class HelloServer {
5      public static void main(String[] args) {
6          RemoteHelloImpl hello = null;
7
8          try { // Create the remote object.
9              hello = new RemoteHelloImpl();
10         } catch (RemoteException e) {
11             // Handle exception...
12         }
13
14         try { // Create an RMI registry.
15             LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
16         } catch (RemoteException e) {
17             // Handle exception...
18         }
19
20         try { // Bind the object in the registry.
21             String url = "//" + getHostName() + "/hello";
22             Naming.bind(url, hello);
23         } catch (AlreadyBoundException e1) {
24             // Handle exception...
25         } catch (java.net.MalformedURLException e2) {
26             // Handle exception...
27         } catch (RemoteException e3) {
28             // Handle exception...
29         }
30     }
31 }
```

Figure 2.6: Code for an RMI server program that uses the RemoteHello class.

The server code that creates and exports the remote object is shown in Figure 2.6. RMI objects can be registered in a registry, using a URL as identifier. This allows the client side to find the remote object. It can do a lookup operation using the same URL.

The code for the client side is shown in Figure 2.7. The client does a lookup to find the remote object, and gets a *stub* to the remote object as a result. The stub implements the RemoteHello interface, which allows the client to invoke the *sayHello* method on the stub. On the server side, a so-called *skeleton* unmarshalls the invocation, and forwards it to the remote object. The stubs and skeletons are generated by the RMI stub compiler, *rmic*.

The example in Figure 2.7 shows why RMI is more flexible than RPC, because it uses polymorphism. The *sayHello* method is invoked with a *String* ("greetings" in the example) as parameter, while the formal type of the parameter is *Object*, as is specified in the *RemoteHello* interface (Figure 2.4) and the implementation (Figure 2.5). In this case, the server does not have to do anything special, as the *String* class is known. However, the *sayHello* method can be invoked with an arbitrary object, which type is potentially

```java
1  import java.rmi.*;
2
3  class HelloClient {
4      public static void main(String argv[]) {
5          RemoteHello hello = null;
6          int result = 0;
7
8          try { // Lookup the remote object, and get a stub back.
9              String url = "//" + getServerHostName() + "/hello"
10             hello = (RemoteHello) Naming.lookup(url);
11         } catch (Exception e) {
12             // Handle exception...
13         }
14
15         try { // Do an RMI to the server.
16             result = hello.sayHello("greetings"); // The RMI.
17         } catch (RemoteException e) {
18             // Handle exception...
19         }
20
21         System.out.println("result = " + result);
22     }
23 }
```

Figure 2.7: Code for an RMI client program that invokes a method on the RemoteHel-loInterface.
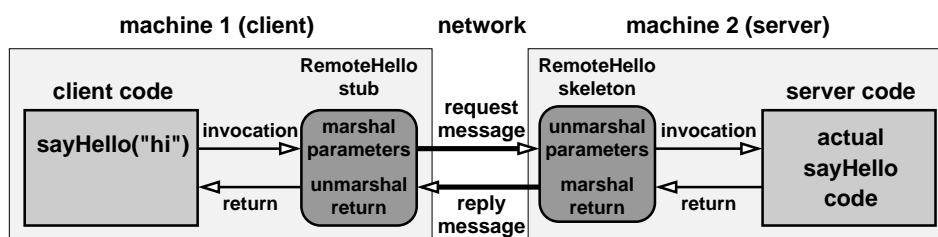
Figure 2.8: Remote invocation of the *sayHello* method.

unknown at the server side. If the type is unknown, the server sets up a HTTP connection to the client to download the bytecode for the class, which is then dynamically loaded into the server.

Figure 2.8 shows what happens when the client invokes the *sayHello* method on the stub. The stub converts the parameters to the method invocation to a stream of bytes that is sent over the network. Java's serialization mechanism is used to implement this. The RMI request is sent to the machine where the remote object resides (normally using TCP/IP), where the message arrives at the skeleton. The skeleton unmarshalls the parameter data, and forwards the call to the remote object. A new thread is created (or a waiting thread out of a pre-started pool is reused) to execute the remote method. This way, multiple

RMIs can be handled simultaneously. Furthermore, the user code in the remote object is allowed to block or wait for other RMIs. When the method invocation is finished, the skeleton marshalls the return value, and sends it back to the stub, which in turn deserializes the data again. Finally, the stub returns the result to the client code that initiated the RMI. Exceptions are supported by RMI. When the remote code throws an exception, it is serialized just as a normal return value, and re-thrown at the client side. As can be seen in Figure 2.7, the use of RMI is fairly transparent. RMI parameters are passed by value (because of the serialization), and RMIs can throw a *RemoteException*, but other than that, an RMI looks exactly the same as a normal invocation. It is important to note that RMIs can never be executed asynchronously, even when the return type is *void*, because RMIs can throw an exception.

In Chapter 3 we present a detailed performance analysis of Java RMI, and we present a highly optimized RMI implementation we created. A case study where we try to use Java RMI for parallel programming is presented in Chapter 4. Related work on RMI is discussed in more detail in Chapter 3.

2.3.4   Java DSMs

Java has a well-designed memory consistency model [68] that allows threads to be implemented efficiently, even on distributed shared memory systems (DSMs). There are many projects that try to run multithreaded Java programs on a cluster of workstations. Java/DSM [176] and Hu et al. [76] implement a JVM on top of the TreadMarks DSM [87]. No explicit communication is necessary, since all communication is handled by the underlying DSM. This has the advantage of transparency, but it does not allow the programmer to make decisions about the data placement and communication of the program. No performance data for Java/DSM were available to us. DOSA [76] is a DSM system for Java based on TreadMarks that allows more efficient fine-grained sharing. Hyperion [4, 114] tries to execute multithreaded shared-memory Java programs on a distributed-memory machine. It caches objects in a local working memory, which is allowed by the Java memory model. The cached objects are flushed back to their original locations (main memory) at the entry and exit of a `synchronized` statement. cJVM is another Java system that tries to hide distribution and provides a single system image [7]. Jackal is an all-software fine-grained DSM for Java based on the Manta compiler [165].

It is desirable to extend DSM systems to work on the grid, as many programmers are familiar with the multithreading paradigm. Wide-area DSM systems should provide what is sometimes called *grid shared memory*. As explained in more detail in [57] (in Section 8.2), such an extension is complicated due to the heterogenous nature of the grid. Different nodes can have different page sizes and data representation (e.g., word length, byte ordering and data alignment). Therefore, the DSM communication layer should take care of these differences between hosts [26, 177]. This, in combination with high wide-area latencies, makes the implementation of efficient DSM systems on the grid challenging. Some projects have tried to create wide-area DSM systems, based on TreadMarks, but the performance of these systems is not (yet) adequate [6]. To our knowledge, there are currently no Java wide-area DSM systems.

2.4 Other Projects using Java for Parallel Programming

Many other projects for parallel programming in Java exist.[5] Titanium [175] is a Java-based language for high-performance parallel scientific computing. It extends the Java language with features like immutable classes, fast multidimensional array access, and an explicitly parallel SPMD model of communication. The Titanium compiler translates Titanium into C. It is built on the Split-C/Active Messages back-end.

The JavaParty system [131] is designed to ease parallel programming in Java. In particular, its goal is to run multithreaded programs with as little change as possible on a workstation cluster. It allows the methods of a class to be invoked remotely by adding a `remote` keyword to the class declaration, removes the need for elaborate exception catching of remote method invocations, and allows objects and threads to be created remotely. Manta optionally allows a similar programming model, but it also supports the standard RMI programming model. Our system is designed from scratch for high performance. JavaParty originally was implemented on top of Sun RMI, and thus suffered from the same performance problem as Sun RMI. The current implementation of JavaParty uses KaRMI [123]. KaRMI and its performance is examined in more detail in Section 7.3.

Spar/Java is a data parallel and task parallel programming language for semiautomatic parallel programming [137], but is intended for data-parallel applications. Spar/Java does not support threads or RMI. The Do! project tries to ease parallel programming in Java using parallel and distributed frameworks [99]. Ajents is a Java system that supports object migration [79]. Also, several Java-based programming systems exist for developing wide-area grid computing applications [1, 14, 120].

An alternative for parallel programming in Java is to use MPI instead of RMI. Several MPI bindings for Java exist [33, 67, 83]. This approach has the advantage that many programmers are familiar with MPI and that MPI supports a richer set of communication styles than RMI, in particular asynchronous and collective communication. However, the MPI message-passing style of communication is difficult to integrate cleanly with Java's object-oriented model. MPI assumes an SPMD programming model that is quite different from Java's multithreading model. Also, current MPI bindings for Java suffer from the same performance problem as most RMI implementations: the high overhead of serialization and the Java Native Interface (JNI). For example, for the Java-MPI system described in [66], the latency for calling MPI from Java is 119 μs higher than calling MPI from C (346 versus 227 μs, measured on an SP2). CCJ [122] is an implementation of a MPI-like interface on top of Java RMI. It can be used with any JVM, but suffers from the performance problems in RMI. When compiled with Manta, the performance is comparable to calling native MPI libraries from Java, due to Manta's efficient RMI implementation.

IceT [69] also uses message passing instead of RMI. It enables users to share Java Virtual Machines across a network. A user can upload a class to another virtual machine using a PVM-like interface. By explicitly calling *send* and *receive* statements, work can be distributed among multiple JVMs. IceT thus provides separate communication primitives, whereas RMI (and Manta) use object invocation for communication.

---

[5]See for example the JavaGrande Web page at *http://www.javagrande.org/.*
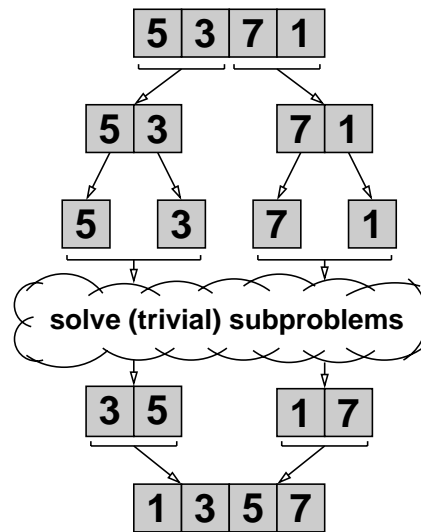
Figure 2.9: A divide-and-conquer sorting algorithm: merge sort.

The above Java systems are designed for single-level ("flat") parallel machines. In contrast, the Manta system described in this thesis is designed for hierarchical systems and uses different communication protocols for local and wide area networks. It uses a highly optimized RMI implementation, which is particularly effective for local communication.

2.5   Divide and Conquer

Divide-and-conquer algorithms work by recursively dividing a problem into smaller subproblems. This recursive subdivision continues until the remaining subproblem becomes trivial to solve. After solving subproblems, the results of these subproblems are recursively recombined again until the final solution is assembled. This process is shown in Figure 2.9 for the merge-sort algorithm. Merge sort works by recursively splitting the array that has to be sorted into two parts. This subdivision goes on until the trivial (solved) subproblem is reached: an array of one element. Next, the sub results are combined again, while keeping the arrays sorted. This goes on until the entire problem is solved and the original array is sorted. Computations that use the divide-and-conquer model include geometry procedures, sorting methods, search algorithms, data classification codes, n-body simulations and data-parallel numerical programs [174].

Divide-and-conquer applications may be parallelized by letting different processors solve different subproblems. These subproblems are often called *jobs* in this context. Generated jobs are transferred between processors to balance the load in the computation. An important characteristic of the divide-and-conquer model is that it uses local synchronization between spawner and spawnee (using a special synchronization statement), but

no global synchronization. Replicated objects can be used when some parts of the application need global synchronization.

The divide-and-conquer model lends itself well for hierarchically-structured systems, because tasks are created by recursive subdivision. This leads to a hierarchical task graph, which can be executed with excellent communication locality, especially on hierarchical platforms. Of course, there are many kinds of applications that do not lend themselves well to a divide-and-conquer algorithm. However, we believe the class of divide-and-conquer algorithms to be sufficiently large to justify its deployment for hierarchical wide-area systems.

Nearly all existing parallel programming systems for grids either assume "trivial" parallelism (i.e., the overall task can be split into independent jobs) or master-slave parallelism. We regard divide-and-conquer as a generalization of master-slave parallelism, essentially allowing a slave to recursively partition the work further. Alternatively, one could say that master-slave parallelism is divide-and-conquer with only one level of recursion. So, our approach is more flexible than what is supported by other systems that are currently in use.

2.6 Load Balancing

Parallel computing involves breaking up problems into parts which are solved concurrently. In some cases, the distribution of work and data is straightforward, or the optimal distribution can be calculated beforehand. For many programs, however, this is not feasible, because the execution times and communication requirements of tasks are not known beforehand. A good distribution of communication and calculation is necessary to avoid idle processors, and thus suboptimal performance. However, finding the optimal distribution of work is a hard problem, in fact, it is NP-complete. When the calculation and communication requirements of subtasks are not known beforehand, the distribution of the work has to be computed *during* the parallel run, and thus can form an important source of overhead. Because finding an optimal solution is NP-complete, many algorithms only approximate the optimal solution. Fox et al. [59], in Chapter 11, give a classification of load balancing strategies, specifying four different types:

1. *By Inspection*: The load-balancing strategy can be determined by inspection, such as with a rectangular lattice of grid points split into smaller rectangles, so that the load balancing problem is solved before the program is written.

2. *Static*: The optimization is nontrivial, but can be computed beforehand on a sequential machine.

3. *Quasi-Dynamic*: The circumstances determining the optimal balance change during program execution, but discretely and infrequently. Because the change is discrete, the load-balancing problem and its solution remain the same until the next change. If the changes are infrequent enough, any savings made in the subsequent computation make up for the time spent solving the load-balancing problem. The difference between this and the static case is that the load balancing is done in parallel to avoid a sequential bottleneck.

4. *Dynamic*: The circumstances determining the optimal balance change frequently or continuously during execution, so that the cost of the load balancing calculation should be minimized in addition to optimizing the splitting of the actual calculation.

In this thesis, we use several applications that use *static* load balancing (e.g., SOR and ASP) in combination with Java RMI. We also use *quasi-dynamic* load balancing for several RMI applications (e.g., Barnes-Hut). Performance results on a local cluster and on a wide-area system are given in Chapters 3 and 4.

In general, when applications target the grid, *quasi-dynamic* or *dynamic* load balancing schemes are needed, because execution and communication speeds can change dramatically over time. Satin supports fine-grained irregular applications where the communication and execution requirements of subtasks are unknown, and thus needs *dynamic* load balancing schemes to achieve good performance. Because Satin also targets the grid, the use of efficient *dynamic* load balancing algorithms is even more critical, due to the dynamic nature of communication performance.

We can split the fourth category (*dynamic*) further into two classes: *synchronous* and *asynchronous*. With synchronous load balancing, computation phases are interleaved with load balancing phases. At certain points, all processors synchronize and start a load balancing phase, where the new work distribution is calculated and work is transferred. Global (system load) information can be used to compute the new distribution, although decentralized schemes are also possible. Asynchronous load balancing does not have phases, and in essence, the load balancing algorithm runs concurrently with the actual application itself. On grids, the latter is preferable, as global synchronization is potentially expensive due to the large WAN latencies. Therefore, Satin uses *asynchronous*, *dynamic* load balancing mechanisms.

Currently, many grid systems use either a centralized load balancing algorithm (e.g., Javelin 2.0 [121]), hierarchical load balancing (e.g., Atlas [14] and Javelin 3 [120]), or random stealing to balance the load in parallel computations. In this thesis, we show that these methods are suboptimal in grid environments, because network and compute performance are very dynamic, and because there is a large performance gap between local and wide-area communication. Random stealing, although proven to be optimal when used on homogeneous hardware, is inefficient for grid applications, because changes in link speed and compute power change the stochastic properties of the algorithm (because more time is spent on transferring tasks over a slow link). The novel algorithms described in this thesis therefore adapt the stochastic properties of work stealing to differences in link speed and compute power (see Section 6.5.2). Related work on load balancing is discussed in Section 6.6.

2.7   Application Programs Used in this Thesis

We investigate the performance of Sun RMI and Manta RMI with six different application kernels (ASP, SOR, Radix, FFT, Water and Barnes) in Chapter 3. We present two additional applications (TSP and IDA*) in Chapter 4, and use them in a case study that evaluates the use of RMI for wide-area parallel programming. Furthermore, we evaluate Satin's performance using twelve application kernels in Chapters 5, 6 and 8. Two of the

applications overlap (TSP and IDA*), and were also used to evaluate RMI. In Chapter 8, we use the additional Satin application Awari to analyze the mechanism for speculative parallelism. We will describe the algorithms used in the application kernels below.

2.7.1   Red/black Successive Over-Relaxation (RMI)

Successive Over-Relaxation is an iterative algorithm for solving Laplace equations on a grid data structure. The sequential algorithm works as follows. For all non-boundary points of the grid, SOR first calculates the average value of the four neighbors of the point:

$$av(r,c) = \frac{g[r+1,c+1] + g[r+1,c-1] + g[r-1,c+1] + g[r-1,c-1]}{4}$$

Then the new value of the point is determined using the following correction:

$$g_{new}[r,c] = g[r,c] + \omega(av(r,c) - g[r,c])$$

ω is known as the *relaxation parameter* and a suitable value can be calculated in the following way:

$$\omega = \frac{2}{1 + \sqrt{1 - \frac{1}{4}(\cos\frac{\pi}{totalcolumns} + \cos\frac{\pi}{totalrows})^2}}$$

The entire process terminates if during the last iteration no point in the grid has changed more than a certain (small) quantity. The parallel implementation of SOR is based on the Red/Black SOR algorithm. The grid is treated as a checkerboard and each iteration is split into two phases, red and black. During the red phase only the red points of the grid are updated. Red points only have black neighbors and no black points are changed during the red phase. During the black phase, the black points are updated in a similar way. Using the Red/Black SOR algorithm the grid can be partitioned among the available processors, each processor receiving a number of adjacent rows. All processors can now update different points of the same color in parallel. Before a processor starts the update of a certain color, it exchanges the border points of the opposite color with its neighbor. After each iteration, the processors must decide if the calculation must continue. Each processor determines if it wants to continue and sends its vote to the first processor. The calculation is terminated only if all the processors agree to stop. This means a processor may continue the calculation after the termination condition is reached locally. Pseudo code and wide-area optimizations for SOR are presented in Section 4.2.1.

The original version of SOR was written by Henri Bal in Orca. Peter Dozy and Mirjam Bakker and Aske Plaat implemented several wide-area optimizations in the Orca version [132]. The author translated this version to Java and converted the program to use RMI and threads (to simulate asynchronous messages over WAN links).

2.7.2   All-pairs Shortest Paths (RMI)

The All-pairs Shortest Paths (ASP) program computes the shortest path between any pair
of nodes in a graph, using a parallel version of Floyd's algorithm. The program uses a
distance matrix that is divided row-wise among the available processors. At the begin-
ning of iteration k, all processors need the value of the kth row of the matrix. The most
efficient method for expressing this communication pattern would be to let the processor
containing this row (called the owner) broadcast it to all the others. Unfortunately, Java
RMI does not support broadcasting, so this cannot be expressed directly in Java. Instead,
we simulate the broadcast with a spanning tree algorithm implemented using RMIs and
threads. Pseudo code and wide-area optimizations for ASP are discussed in Section 4.2.2.

For this thesis, the author translated a wide-area optimized ASP version in the Orca
language by Aske Plaat [132] into Java, and converted it to use RMI and threads. The
original ASP application was written by Henri Bal. The broadcast implementation uses
pre-started threads from a thread pool to simulate asynchronous WAN messages. Wide-
area optimizations for ASP are described in more detail in Section 4.2.2.

2.7.3   Radix Sort (RMI)

Radix sort is a histogram-based parallel sort program from the SPLASH-2 suite [173].
The Java-RMI version of radix sort was implemented by Monique Dewanchand and is
described in more detail in [44]. The program repeatedly performs a local sort phase
(without communication) followed by a histogram merge phase. The merge phase uses
combining-tree communication to transfer histogram information. After this merge phase,
the program moves some of the keys between processors, which also requires RMIs.

2.7.4   Fast Fourier Transform (RMI)

FFT is a single dimensional Fast Fourier Transform program based on the SPLASH-
2 code which was rewritten in Java by Ronald Blankendaal and is described in more
detail in [44]. The matrix is partitioned row-wise among the different processors. The
communication pattern of FFT is a personalized all-to-all exchange, implemented using
an RMI between every pair of machines.

2.7.5   Water (RMI)

Water is another SPLASH application that was rewritten in Java by Monique Dewanc-
hand and is described in more detail in [44]. This N-body simulation is parallelized by
distributing the bodies (molecules) among the processors. Communication is primarily
required to compute interactions with bodies assigned to remote machines. Our Java pro-
gram uses message combining to obtain high performance: each processor receives all
bodies it needs from another machine using a single RMI. After each operation, updates
are also sent using one RMI per destination machine. Since Water uses an $O(N^2)$ algo-
rithm and we optimized communication, the relative communication overhead is low.

2.7.6 Barnes-Hut (RMI)

Barnes-Hut is an $O(N \log N)$ N-body simulation. Martijn Thieme wrote a Java program based on the code by Blackston and Suel [21]. This code is optimized for distributed-memory architectures. The Java version of the application is described in more detail in [163]. Instead of finding out at run time which bodies are needed to compute an interaction (as in the SPLASH-2 version), this code pre-computes where bodies are needed, and sends them in one collective communication phase before the actual computation starts. In this way, no stalls occur in the computation phase [21].

2.7.7 Traveling Salesperson Problem (RMI, Satin)

The Traveling Salesperson Problem (TSP) computes the shortest path for a salesperson to visit all cities in a given set exactly once, starting in one specific city. We use a branch-and-bound algorithm, which can prune a large part of the search space. When the length of the current partial route and the minimal length of a path connecting the remaining cities together are already longer than the best known solution, the path can be pruned. As explained in [12], the program can be made deterministic by initializing the minimum bound to the best solution. For performance comparisons we use the deterministic version. However, to demonstrate the use of replicated objects and aborts in combination with Satin, we also provide some measurements using the nondeterministic version. In the remainder of this thesis, we explicitly state whether the deterministic version or the nondeterministic version is used.

The program is parallelized by distributing the search space over the different processors. Because the algorithm performs pruning, however, the amount of computation needed for each sub-space is not known in advance and varies between different parts of the search space. Therefore, the load must be balanced dynamically. For the Satin version of TSP, load distribution is done automatically by the Satin runtime system, by letting idle nodes steal work from nodes that still have work. The global minimum is implemented as a replicated object in the Satin version. The RMI version implements a load balancing mechanism at the application level. In single-cluster systems, load imbalance can easily be minimized using a centralized job queue. In a wide-area system, this would generate much wide-area communication. As wide-area optimization we implemented one job queue per cluster.

The RMI version also replicates the best known path length at the application level. Each worker contains a copy of the minimum value. If a worker finds a better complete route, the program does an RMI to all other peer workers to update their copies. To allow these RMIs, the objects that contain the minimum values are declared as remote objects.

The TSP version used in this thesis is a translation done by the author, from the original code in the Orca language by Henri Bal. Next, we applied the wide area optimizations described in [132]. The code was translated into Java, and converted to use RMI and threads. The Satin version (written from scratch by the author) does not need any wide-area optimizations or load balancing code, because the Satin runtime system implements these. As a result, the Satin version of the program is much shorter.
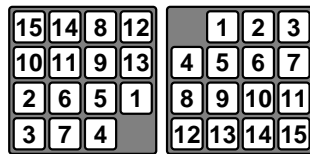
Figure 2.10: A scrambled (left) and the target position (right) of the 15-puzzle.

2.7.8   Iterative Deepening A* (RMI, Satin)

Iterative Deepening A* (IDA) is another combinatorial search algorithm, based on repeated depth-first searches. IDA* tries to find all shortest solution paths to a given problem by doing a depth-first search up to a certain maximum depth. If the search fails, it is repeated with a higher search depth, until a solution is found. The search depth is initialized to a lower bound of the solution. The algorithm thus performs repeated depth-first searches. Like branch-and-bound, IDA* uses pruning to avoid searching useless branches.

We have written an IDA* program in Java that solves the 15-puzzle (the sliding tile puzzle). The program finds all shortest solutions from a scrambled position to the target position, see Figure 2.7.8. IDA* is parallelized by searching different parts of the search tree concurrently. From each game position, three new boards are generated by sliding the blank in a different direction (the fourth direction would just undo the last step). In the Satin version, the three generated boards are examined in parallel by spawning the searching method.

The RMI program implements a load balancing mechanism at the application level. We use a mechanism that is based on random work stealing. We implemented one wide-area optimization: to avoid wide-area communication for work stealing, each machine first tries to steal jobs from machines in its own cluster. Only if that fails, the work queues of remote clusters are accessed. In each case, the same mechanism (RMI) is used to fetch work, so this heuristic is easy to express in Java.

The Satin version of the program does not need distributed termination detection, because of the divide-and-conquer model. When a spawned method is finished, the Satin runtime system notifies the spawner of the work. Due to the recursive model, the root node in the spawn tree will be notified when all work is finished. This is efficient, because the notification messages are sent asynchronously. The RMI version uses a straightforward centralized termination detection algorithm. It does not use notification messages when a job is finished, because this would be expensive with the synchronous RMI model.

The program can be made deterministic by not stopping the search when the first solution is found, but instead continuing until *all* solutions are found. For performance comparisons we use the deterministic version.

The IDA* version used in this thesis is a translation done by the author from the code in the Orca language by Aske Plaat [132]. While the RMI version uses RMI and threads, the Satin version does not need any wide-area optimizations or load-balancing code, because the Satin runtime system implements these. As a result, the Satin version of the program is much shorter.
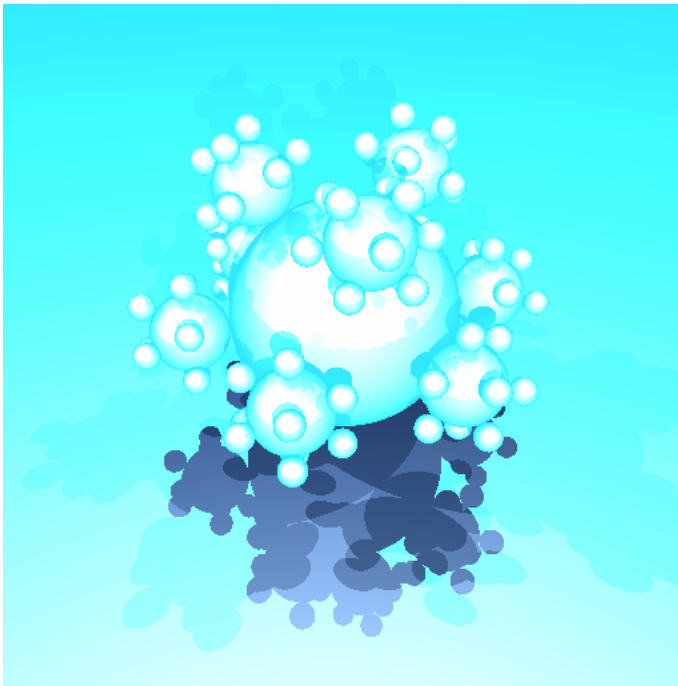
Figure 2.11: Scene computed by the raytracer.

2.7.9   Fibonacci and Fibonacci Threshold (Satin)

The Fibonacci program calculates the Fibonacci numbers, and is a typical example of a divide-and-conquer program. The code for Fibonacci is shown in Figure 5.1. Note that this is a benchmark, and *not* a suitable algorithm for efficiently calculating the Fibonacci numbers (the iterative algorithm that calculates the Fibonacci numbers has complexity O(n)). A well known optimization in parallel divide-and-conquer programs is to make use of a threshold on the number of spawns. When this threshold is reached, work is executed sequentially. This approach can easily be programmed using Satin, and will be investigated in more detail in Section 5.3.1.

2.7.10   The Knapsack Problem (Satin)

Here, the goal is to find a set of items, each with a weight w and a value v such that the total value is maximal, while not exceeding a fixed weight limit. The problem for n items is recursively divided into two subproblems for $n - 1$ items, one with the missing item put into the knapsack and one without it. The original program was written by Thilo Kielmann [61] and has been converted from C to Java by the author.
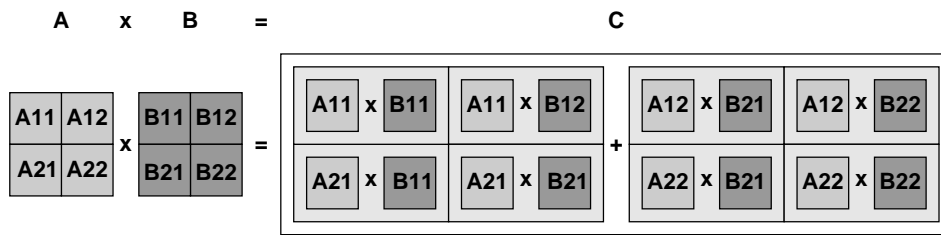
Figure 2.12: Multiplication of the matrices A and B by dividing into quadrants.

2.7.11   Prime Factorization (Satin)

This application is an algorithm for the decomposition of positive integer numbers into prime factors, also known as integer factorization [40]. The algorithm implemented here is not one of the popular heuristic ones. Instead, it finds all prime factors of a given number N in the interval ranging from n_1 to n_2. To achieve this, it divides the given interval into two of half the size and tests their elements respectively. The original program was written by Thilo Kielmann [61] and has been converted from C to Java by the author.

2.7.12   Raytracer (Satin)

The raytracer application computes a picture using an abstract description of a scene. The application is parallelized by recursively dividing the picture up to the pixel level. Next, the correct color for that pixel is calculated and the image is reassembled. The scene that we use in this thesis is shown in Figure 2.11. The sequential Java version was implemented by Ronald Veldema and parallelized by the author.

2.7.13   Matrix multiplication (Satin)

This application multiplies two matrices A and B, and stores the result in a new matrix C, by recursively dividing A and B into four quadrants each, using the algorithm illustrated by Figure 2.12. The recursion stops when a certain threshold is reached. We set the threshold to blocks of 48x48 doubles in this thesis. The resulting blocks are multiplied using a sequential version of matrix multiplication that steps two rows and columns at a time. Another optimization is that the results are accumulated into C, not just set into C. This works well here because Java array elements are created with all zero values.

Aske Plaat translated a Cilk program for matrix multiplication (written by Bobby Blumofe) to a Java version that uses Java threads for parallelism. The author modified this version to use Satin's constructs for parallelism, and replaced the sequential algorithm by the more efficient version used by Doug Lea in his Java Fork/Join Framework [100].

2.7.14 Adaptive Numerical Integration (Satin)

This application is an algorithm for adaptive numerical integration [3], i.e., computing $\int_a^b f(x)dx$ for a function $f(x)$ in the interval between the points a and b. The basic idea of the algorithm is that $f(x)$ is replaced by a straight line from $(a, f(a))$ to $(b, f(b))$ and the integral is approximated by computing the area bounded by the resulting trapezoid. This process is recursively continued for two subintervals as long as the area differs significantly from the sum of the areas of the subintervals' trapezoids. The algorithm is adaptive in the sense that the partitioning into subintervals depends on the shape of the function considered. The original program was written by Thilo Kielmann [61] and has been converted from C to Java by the author.

2.7.15 N Choose K (Satin)

This kernel computes the binomial coefficient, often also referred to as "n over k" or "n choose k". The binomial coefficient is defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

The original program was written by Thilo Kielmann [61] in C, and has been converted from C to Java by the author.

2.7.16 The Set Covering Problem (Satin)

This applications implements an exact algorithm for the set-covering problem [40]. The task is to find a minimal set of subsets of a set S which covers the properties of all elements of the set S. The original program was written by Thilo Kielmann [61] and has been converted from C to Java by the author.

2.7.17 N Queens (Satin)

The N-Queens benchmark solves the combinatorially hard problem of placing N queens on a chess board such that no queen attacks an other. Recursive search is used to find a solution.

The program can be made deterministic by not stopping the search when the first solution is found, but instead continuing the search until *all* solutions are found. For performance comparisons we use the deterministic version. However, to demonstrate the use of replicated objects and aborts in combination with Satin, we also provide some measurements using the nondeterministic version. Whenever N-Queens is used in the remainder of this thesis, we explicitly state whether the deterministic version or the nondeterministic version is used.

The original program we used was taken from the examples in Doug Lea's Java Fork/Join Framework [100]. The author rewrote the program to use Satin's primitives. Furthermore, the code was optimized a bit further.

2.7.18   Awari (Satin)

Awari [43] (also known as Wari, Owari, Awale, Awele, and Ayo) is an ancient, well-known game (a Mancala variant) that originates from Africa, and is played worldwide now. Awari is played using 12 "pits" and 48 seeds or stones. The aim is for one player to capture more than half the number of seeds. Recently, the game was solved by Romein and Bal [145].

The author wrote a Satin Awari implementation from scratch, together with John Romein. Our implementation uses the MTD(f) [133] algorithm to do a forward search through the possible game states. A transposition table [153] is used to avoid searching identical positions multiple times. The parallel version speculatively searches multiple states in parallel, and replicates (a part of) the transposition table to avoid search overhead. Message combining is used to aggregate multiple transposition table updates into a singe network message, to avoid excessive communication. Pseudo code for the algorithm we used is presented in Chapter 8 (see Figures 8.16 and 8.17).

Chapter 3

The Manta RMI System

```
He who has not first laid his foundations may be able with great
ability to lay them afterwards, but they will be laid with trouble
to the architect and danger to the building.

- Niccolo Machiavelli
```

Remote procedure calls (RPCs) are proposed as a building block for grid applications in [101, 102, 150]. RMI [157] is an object oriented form of RPC, so it should in theory also be suitable for grid programming. The RMI programming model, and the differences between RMI and RPC, are described in more detail in Chapter 2. To use Java for high performance (grid) computing, two issues are important. First, efficient sequential program execution is imperative. In this thesis, we use a native compiler, named Manta, to achieve good sequential performance. The performance of Manta, and optimizations for sequential speed are described in more detail in [165]. Second, efficient communication primitives are important. These communication primitives and their performance are the subject of this chapter.

We assume that the grid is hierarchically structured (i.e., it consists of connected clusters of workstations). Therefore, the performance of an application on a single cluster is important, as clusters will be an important building block of the grid. As we will see in Chapter 4, it is desirable to keep communication within a cluster as much as possible, because wide-area communication is expensive. Also, for some applications it is possible to reduce wide-area communication at the cost of increased local communication (see also Chapter 4 and 6). This makes the performance of local communication primitives even more important. Therefore, this chapter focuses on cluster-local communication in Java.

For shared-memory machines, Java offers a familiar multithreading paradigm. For distributed-memory machines, such as clusters of workstations, Java provides Remote Method Invocation (RMI), which is an object-oriented version of RPC. The RMI model offers many advantages for distributed programming, including a seamless integration with Java's object model, heterogeneity, and flexibility [167]. The support for computing on heterogeneous systems extends Java's applicability to grid computing or parallel processing on computational grids [57].

Unfortunately, many existing Java implementations have inferior performance of both sequential code and communication primitives, which is a serious disadvantage for high-performance computing. Much effort is invested in improving sequential code performance by replacing the original bytecode interpretation scheme with just-in-time compilers, native compilers, and specialized hardware [31, 92, 119, 135]. The communication overhead of RMI implementations, however, remains a major weakness. RMI is designed for client/server programming in distributed (Web-based) systems, where network latencies in the order of several milliseconds are typical. On more tightly coupled parallel machines, such latencies are unacceptable. On our Pentium Pro/Myrinet cluster, for example, Sun's JDK 1.2 (Java Developer Kit) implementation of RMI obtains a *null-RMI latency* (i.e., the round-trip time of an RMI without parameters or a return value) of 1316 μs, compared to 31 μs for a user-level Remote Procedure Call protocol in C.

Part of this large overhead is caused by inefficiencies in the JDK implementation of RMI, which is built on a hierarchy of stream classes that copy data and call virtual methods. Serialization of method arguments (i.e., converting them to arrays of bytes) is implemented by recursively inspecting object types until primitive types are reached, and then invoking the primitive serializers. All of this is performed at run time, for each remote invocation. In addition, RMI is implemented on top of IP sockets, which adds kernel overhead and context switches to the critical path.

Besides inefficiencies in the JDK implementation of RMI, a second reason for the slowness of RMI is the difference between the RPC and RMI models. Java's RMI model is designed for flexibility and inter operability. Unlike RPC, it allows classes unknown at compile time to be exchanged between a client and a server and to be downloaded into a running program. In Java, an actual parameter object in an RMI can be of a subclass of the class of the method's formal parameter. In (polymorphic) object-oriented languages, the *dynamic* type of the parameter-object (the subclass) is used by the method, not the static type of the formal parameter. When the subclass is not yet known to the receiver, it has to be fetched from a file or HTTP server and be downloaded into the receiver. This high level of flexibility is the key distinction between RMI and RPC [167]. RPC systems simply use the static type of the formal parameter (thereby type-converting the actual parameter), and thus lack support for polymorphism and break the object-oriented model. An example of polymorphic RMI is shown in Section 2.3.3.

Our goal is to obtain the efficiency of RPC *and* the flexibility of Java's RMI. This chapter discusses the Manta compiler-based Java system, which has been designed from scratch to implement RMI efficiently. Manta replaces Sun's runtime protocol processing as much as possible by compile-time analysis. Manta uses a native compiler to generate efficient sequential code and specialized serialization routines for serializable argument classes. The generated serializers avoid using reflection (runtime type inspection) and data copying. Also, Manta uses an efficient wire format and sends type descriptors for argument classes only once per destination machine, instead of once for every RMI. In this way, almost all of the protocol overhead has been pushed to compile time, off the critical path.

The problems with this approach are, however, how to interface with other Java Virtual Machines (JVMs) and how to address dynamic class loading. Both are required to support inter operability and polymorphism. To inter operate with JVMs, Manta supports the Sun

RMI and serialization protocol in addition to its own protocol. Dynamic class loading is supported by compiling methods and generating serializers at run time. Furthermore, bytecodes that are used by the application are stored with the executable program to allow the exchange of classes with JVMs.

The general strategy of Manta is to make the frequent case fast. Since Manta is designed for parallel processing, we assume that the frequent case is communication between Manta processes, running on different nodes within a cluster, for example. Manta supports the infrequent case (communication with JVMs) using a slower approach. The Manta RMI system therefore logically consists of two parts:

- A fast communication protocol that is used only between Manta processes. We call this protocol *Manta RMI*, to emphasize that it delivers the standard RMI programming model to the user, but it can only be used for communication between Manta processes.

- Additional software that makes the Manta RMI system as a whole compatible with standard RMI, so Manta processes can communicate with JVMs.

We refer to the combination of these two parts as the *Manta RMI system*. We use the term *Sun RMI* to refer to the standard RMI protocol as defined in the RMI specification [157]. Note that both Manta RMI and Sun RMI provide the same programming model, but their wire formats are incompatible.

The Manta RMI system thus combines high performance with the flexibility and inter operability of RMI. In a grid computing application [57], for example, some clusters can run our Manta software and communicate internally using the Manta RMI protocol. Other machines may run JVMs, containing, for example, a graphical user interface program. Manta communicates with such machines using the Sun RMI protocol, allowing method invocations between Manta and JVMs. Manta implements almost all other functionality required by the RMI specification, including heterogeneity, multithreading, synchronized methods, and distributed garbage collection. Manta currently does not implement all of Java's security model, as the system is a research vehicle, and not a production system yet. This problem is solved by the Ibis system, which will be described in Chapter 7.

We evaluate the performance of Manta using benchmarks and applications that run on the DAS (see Section 1.9). The time for a null-RMI (without parameters or a return value) of Manta is 35 times lower than that for the Sun JDK 1.2, and only slightly higher than that for a C-based RPC protocol. This high performance is accomplished by pushing almost all of the runtime overhead of RMI to compile time.

We compare the performance of Manta RMI with an optimized implementation of the Sun RMI protocol, compiled with Manta. This implementation is called *Sun compiled*. Using this optimized implementation, we show that current implementations (in the Sun and IBM JDKs) of the Sun RMI protocol are inefficient. More importantly however, we also show that the Sun RMI protocol is inherently inefficient: it *forces* an inefficient implementation.

The contributions of this chapter are the following:

1. we provide a performance analysis and breakdown of Java serialization and RMI and show why Sun RMI implementations are inefficient;

2. we show that the Sun RMI protocol is inherently inefficient, because it forces byte swapping, sends type information multiple times, and makes a zero-copy implementation impossible;

3. using Manta, we demonstrate that serialization and communication in Java (and RMI in particular) can be made almost as efficient as C-based RPC, even on gigabit/s networks;

4. we illustrate several optimizations that have to be implemented to achieve this;

5. we optimize RMI, while maintaining the advantages of RMI over RPC, such as polymorphism; we show how to obtain the *efficiency of RPC* and the *flexibility of RMI* at the same time.

6. we show that it is still possible to inter operate with any JVM, although the wire format of Manta RMI is incompatible with Sun RMI;

7. we present application performance measurements to show that RMI performance can have a large impact on the efficiency of parallel applications.

The outline of the chapter is as follows. In Section 3.1, we give an overview of the Manta RMI system. Next, in Section 3.2, we give an example that uses the Manta and Sun RMI protocols at the same time, to explain the usefulness of our approach. Section 3.3 describes the design and implementation of Manta RMI in detail. The implementation of a Sun RMI system using Manta is described in Section 3.4. In Section 3.5, we give an in-depth analysis of the performance of Manta RMI, and we compare this to the performance of Sun RMI, compiled with the Manta compiler. Section 3.6 briefly describes the impact of the RMI performance at the application level, using six application kernels. We discuss related work in Section 3.7 and give our conclusions in Section 3.8.

3.1   Design Overview of the Manta RMI System

This section discusses the design of the Manta RMI system, which includes the Manta RMI protocol and the software extensions that make Manta compatible with Sun RMI. Manta uses a native compiler to make debugging and profiling more convenient. The latter feature is used extensively throughout this chapter to provide detailed low-level performance measurements. To support inter operability with JVMs the Manta runtime system also has to be able to process bytecode for dynamically loaded classes, and therefore contains a runtime bytecode-to-native compiler.

The Manta RMI system is illustrated in Figure 3.1. The box labeled "Host 2" describes the structure of a Manta process. Such a process contains the executable code for the application and (de)serialization routines, both of which are generated by Manta's native compiler. Manta processes can communicate with each other through the Manta RMI protocol, which has its own wire format (e.g., communication between hosts 1 and 2). A Manta process can communicate with any JVM (the box on the right) through the Sun RMI protocol (i.e., communication between hosts 2 and 3), using the standard RMI
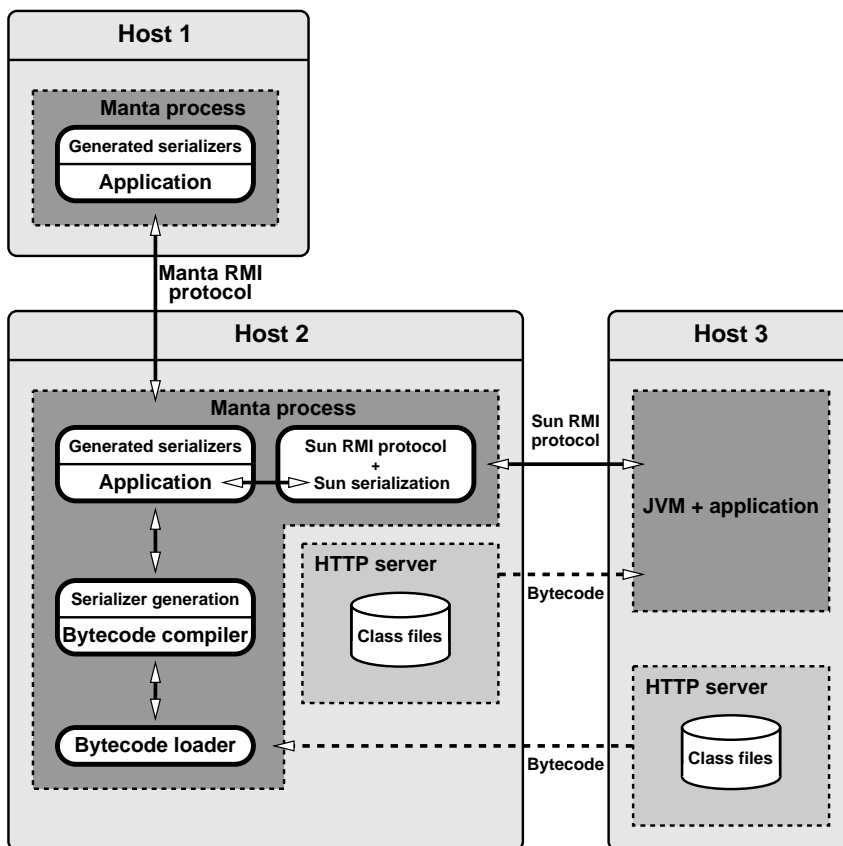
Figure 3.1: Manta/JVM inter operability.

format (i.e., the format defined in Sun's RMI specification). The protocol actually used is determined by the type of the registry (i.e., the RMI name server). The Manta protocol uses its own registry. When a registry stub is created for communicating with the registry, a probe is sent to the host that runs the registry. This probe is only recognized by a Manta registry. If the registry turns out to be a Manta registry, the Manta protocol is used; if it turns out to be a JDK registry, the Sun RMI wire protocol is used.

A Manta-to-Manta RMI is performed with the Manta protocol, which is described in detail in the next section. Manta-to-Manta communication is the common case for high-performance parallel programming, for which our system is optimized. Manta's serialization and deserialization protocols support heterogeneity (RMIs between machines with different byte orderings or alignment properties).

A Manta-to-JVM RMI is performed with a slower protocol that is compatible with Sun's RMI specification and the standard RMI wire format. Manta uses generic routines to (de)serialize the objects to or from the standard format. These routines use runtime

type inspection, similar to Sun's implementation. The routines are written in C, as is all of Manta's runtime system, and execute more efficiently than Sun's implementation, which is partly written in Java and uses expensive Java native interface calls to interface with the runtime system.

To support polymorphism for RMIs between Manta and JVMs, a Manta process is able to handle bytecode from other processes (classes that are exported by a JVM, but that have not been statically compiled into the Manta program). When a Manta process requests bytecode from a remote process, Manta will invoke its bytecode compiler to generate the (de)serialization and marshalling routines and the object code for the methods, as if they were generated by the Manta source code compiler. The dynamically generated object code is linked into the application with the operating system's dynamic linking interface (e.g., `dlopen()`). If a remote process requests bytecode from a Manta process, the JVM bytecode loader retrieves the bytecode for the requested class in the usual way through a shared file system or through an HTTP daemon. RMI does not have separate support for retrieving bytecodes.[1] Sun's `javac` compiler can be used to generate the bytecode at compile time. The bytecode has to be stored at a local web server. Manta's Dynamic bytecode compilation is described in more detail in [112].

The structure of the Manta system is more complicated than that of a JVM. Much of the complexity of implementing Manta efficiently is due to the need to interface a system based on a native-code compiler with a bytecode-based system. The fast communication path in our system, however, is straightforward: the Manta protocol just calls the compiler-generated serialization routines and uses a simple scheme to communicate with other Manta processes. This fast communication path, which is typically taken for communication within a parallel machine or a cluster of workstations, is used in the example below.

3.2   Example of Manta-Sun Inter Operability

Manta's RMI inter operability and dynamic class loading is useful to inter operate with software that runs on a JVM and uses the Sun RMI protocol. This makes it possible to run applications on a large scale system (the grid), consisting of Manta and non-Manta nodes. Another example is a parallel program that generates output that must be visualized. The parallel program is compiled with Manta and uses the Manta RMI protocol. The software for the visualization system to be used, however, may be running on the Sun JDK and may use the Sun RMI protocol. To illustrate this type of inter operability, we implemented a simple example, using a version with a graphical extension of one of our parallel applications (Successive Over-Relaxation (SOR), see Sections 2.7.1 and 4.2.1). In this example, a parallel Manta program inter operates with a simple visualization system that was implemented using Sun RMI, and also dynamically loads bytecode from this visualization system.

The computation is performed by a parallel program that is compiled with Manta and runs on a cluster computer (see Figure 3.2). The output is visualized on the display of a workstation, using a graphical user interface (GUI) application written in Java. The

---

[1] See also *http://sirius.ps.uci.edu/~smichael/rmi.htm*
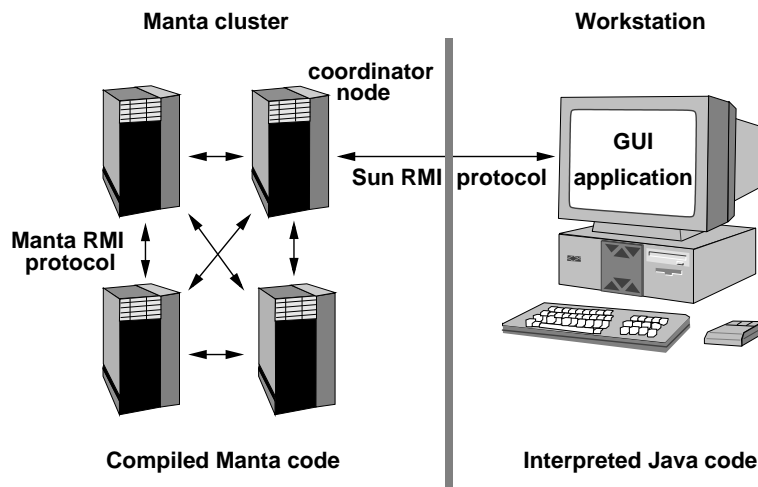
Figure 3.2: Example of Manta's inter operability.

parallel application repeatedly performs one iteration of the SOR algorithm and collects its data (a two-dimensional array) at one node of the cluster, called the *coordinator*. The coordinator passes the result via the Sun RMI protocol to a remote *viewer* object which is part of the GUI application on the workstation. The viewer object creates a window and displays the data. Since the GUI application is running on a Sun JVM, communication between the coordinator and the GUI uses the Sun RMI protocol. The Manta nodes internally use the Manta RMI protocol.

The RMI from the coordinator to the GUI is implemented using a remote *viewer* interface, for which an RMI stub is needed. This stub is not present in the compiled Manta executable, so the coordinator node dynamically retrieves the bytecode for the stub from the code base of the GUI, compiles it using Manta's dynamic bytecode compiler, and then links it into the application. Next, it uses this new stub to present the results to the GUI.

3.3 Manta RMI Implementation

RMI systems can be split into three major components: low-level communication, serialization, and the RMI protocol (stream management and method dispatch). The runtime system for the Manta serialization and Manta RMI protocols are written in C. They were designed to minimize serialization and dispatch overhead, such as copying, buffer management, fragmentation, thread switching, indirect method calls, element-by-element serialization of primitive arrays, and runtime type lookup.

Figure 3.3 gives an overview of the layers in the Manta serialization and Manta RMI and the Sun RMI system. The shaded layers denote statically compiled code, while the white layers are mainly JIT-compiled Java code (although it may contain some native calls). Manta avoids the stream layers of Sun RMI, which are used for serialization.
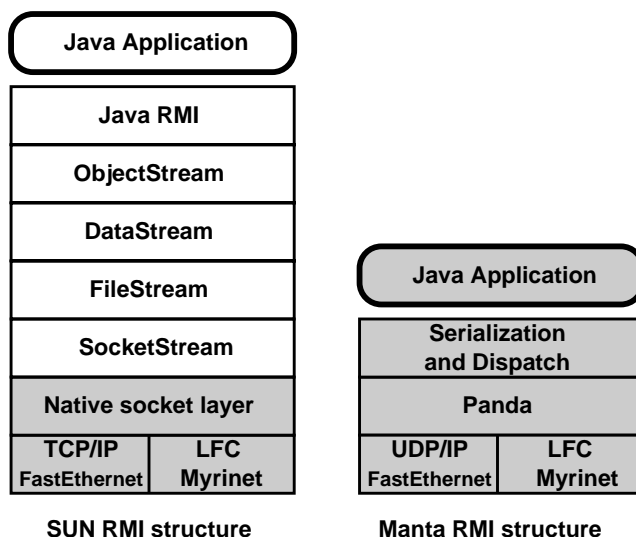
Figure 3.3: Structure of Sun and Manta RMI protocols; shaded layers run compiled code.

Manta does support the programmer's use of Java's stream-classes, but it does not use them for RMI, unless the programmer explicitly writes to them. Instead, RMI parameters are serialized directly into a network buffer. Moreover, in the JDK, these stream layers are written in Java, and their overhead thus depends on the quality of the Java implementation. In Manta, all layers are either implemented as compiled C code or native code that was generated by the Manta compiler. Also, the native code generated by the Manta compiler calls RMI serializers directly, instead of using the slow Java Native Interface. Moreover, Manta uses the efficient, user-level threads and locks which are provided by Panda [147]. The current JDKs use more expensive kernel level locks and threads.

3.3.1   Low-Level Communication

RMI implementations typically are built on top of TCP/IP, which was not designed for parallel processing. Manta uses the Panda communication library, which has efficient implementations on a variety of networks. Panda uses a scatter/gather interface to minimize the number of memory copies, resulting in high throughput. The Manta serialization and RMI protocols use this scatter/gather interface, and thus provide zero-copy communication, both for arrays and objects. To achieve this, the serialization and communication systems are tightly integrated. The measurements that are presented in this chapter were performed on the DAS platform, where Panda uses LFC for low-level communication. Panda and LFC are explained in more detail in Section 1.3.

```
1 class Foo implements java.io.Serializable {
2     int i1, i2;
3     double d;
4     int[] a;
5     Object o;
6     String s;
7     Foo f;
8
9     void foo() { /* foo code */ }
10    void bar() { /* bar code */ }
11 }
```

Figure 3.4: An example serializable class: *Foo*.

3.3.2 The Serialization Protocol

The serialization of method arguments is an important source of overhead in existing RMI implementations. Serialization takes a Java object and converts (serializes) it into a stream of bytes, making a deep copy that includes the referenced sub-objects. The Sun serialization protocol is written in Java and uses reflection to determine the type of each object during run time. RMI implementations use the serialization protocol for converting data that are sent over the network. The process of serializing all arguments of a method is called *marshalling*. Serialization is explained in more detail in Section 2.3.2.

With the Manta protocol, all serialization code is generated by the compiler, avoiding the overhead of reflection. Serialization code for most classes is generated at compile time. Only serialization code for classes which are not locally available is generated at run time, by the bytecode compiler, at marginal extra cost. The overhead of this runtime code generation is incurred only once: when first time the new class is used as an argument to some method invocation. For subsequent uses, the efficient serializer code is available for reuse. The overhead of runtime serializer generation is in the order of a second at worst, depending mostly on whether the Manta bytecode compiler is resident, or whether it has to be paged in. Compiler generation of serialization code is one of the major improvements of Manta over the JDK. It has been used before in Orca [12].

We will explain how Manta serialization works using a toy Java class (called *Foo*), shown in Figure 3.4. The generated (de)serialization pseudo code for *Foo* is shown in Figure 3.5. The code uses Panda's scatter/gather interface. Panda uses I/O-vectors to implement the "gather" side. An I/O-vector is a list of (pointer, size) pairs, and is used to indicate how much data from which memory locations should be sent over the network. The generated serializer uses one *panda_iovector_add* call (line 3) to add a single pointer to all data fields of the object to a Panda I/O-vector at once, without actually copying the data itself. Next, all reference fields in the object are traversed with *writeObject* calls (lines 6–9). Cycles are handled inside *writeObject*. When the reference field points to an object that was not serialized before, *writeObject* calls the correct generated serializer for the referred object.

The serializer stores all fields in the I/O-vector, including the reference fields. The references are not valid on the receiving side, but are used as *opcodes* instead. When

```
1  void serialize_Foo(MarshallStruct *m, class_Foo *self) {
2      /* use zero−copy for fields */
3      panda_iovector_add(m−>iovec, &first_field, total_size_of_Foo);
4
5      /* traverse object fields when non−null */
6      if (self−>a != NULL) writeObject(m, self−>a);
7      if (self−>o != NULL) writeObject(m, self−>o);
8      if (self−>s != NULL) writeObject(m, self−>s);
9      if (self−>f != NULL) writeObject(m, self−>f);
10 }
11
12 void deserialize_Foo(MarshallStruct *m, class_Foo *self) {
13     /* use zero−copy for fields */
14     panda_network_consume(m−>msg, &first_field, total_size_of_Foo);
15
16     /* traverse object fields when non−null */
17     if (self−>a != NULL) self−>a = readObject(m);
18     if (self−>o != NULL) self−>o = readObject(m);
19     if (self−>s != NULL) self−>s = readObject(m);
20     if (self−>f != NULL) self−>f = readObject(m);
21 }
```

Figure 3.5: The generated serialization (pseudo) code for the *Foo* class.

a reference is *null*, the deserializer detects this, and does not read the object. When a reference is non-*null*, the sender uses the pointer as hash value in the cycle check table. On the receiving side, the procedure is similar. First, the data is consumed directly from the network into the object (line 14). Next, the object's reference fields are traversed with *readObject* calls (lines 17–20).

Arrays of primitive types (e.g., *int*, *double*, etc.) are handled specially in Manta. An entry (array length, start of array) is simply added to the I/O-vector, and no copying or byte swapping is performed. Multidimensional arrays and arrays of objects are traversed until the base type is reached.

The process of serializing the *Foo* object is shown in more detail in Figure 3.6. Type information and other opcodes that describe the layout of the stream are saved in a special opcode buffer. The figure shows how the generated serialization code constructs the I/O-vector shown in the lower left part. The I/O-vector only stores pointers to the object data, not the data itself (i.e., no copies are made).

In Manta, all serializable objects have pointers to the serializer and deserializer for that object in their virtual method table (also called vtable). This mechanism is shown in more detail in Figure 3.7. When a particular object is to be serialized, the method pointer is extracted from the method table of the object's *dynamic* type, and then the serializer is invoked. On deserialization, the same procedure is applied.

The fact that the (de)serializer of the dynamic type is called using the object's method table allows further optimization. Consider, for example, a class *Faa*, which extends *Foo*. When an object of class *Faa* is to be serialized, the generated serializer is called via the method table of *Faa*. The serialization code for Faa directly serializes all fields, including
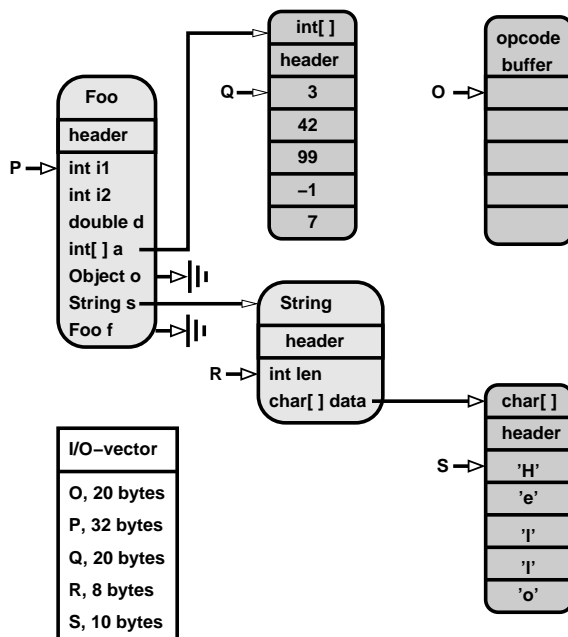
Figure 3.6: Serializing *Foo*.

those of its subclasses. This way, no function calls are required to serialize super classes. When a class to be serialized/deserialized is marked *final*, the cost of the (virtual) function call to the right serializer/deserializer is optimized away, since the correct function pointer can be determined at compile time.

Heterogeneity between little-endian and big-endian machines (or machines with any other byte ordering) is handled in Manta serialization by storing data in the native byte ordering of the serializer, and having the deserializer do the conversion, but only if necessary. Thus, byte swapping between identical machines is avoided. The Manta compiler generates two versions of the deserialization code: one that does byte swapping (from any byte ordering to the byte ordering of the receiver) and one that does not (not shown in Figure 3.5). With Sun serialization, the serializer always converts the data (e.g., the arguments of an RMI call) to the wire format defined by the serialization specification; the deserializer converts the format back to what it requires. If the serializer and deserializer have the same format, but this format is different from the standard serialization format (as is the case on Intel x86 machines!), Sun serialization will do two byte-swap conversions while Manta will not do any byte swapping.

Another important optimization we implemented is related to the creation of the objects during deserialization. Normally, all objects and arrays in Java are initialized to a default value (e.g., *false*, zero or *null*). However, when objects are created during deserialization, the runtime system knows that all array elements and object fields will be
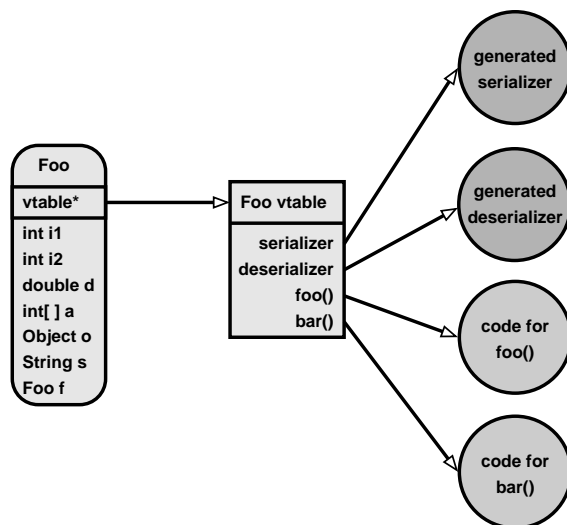
Figure 3.7: The method table for serializable classes.

overwritten with data from the stream. Therefore, we created special versions of *new*, that do not wipe the newly created objects, thus avoiding an expensive *memset* operation. We found that this optimization has a large impact on the throughput (see section 3.5.4).

3.3.3   The Manta RMI Protocol

The Manta compiler also generates the marshalling code for methods that can be called via RMI (i.e., stubs and skeletons). The compiler generates method-specific marshall and unmarshall functions, which (among others) call the generated routines to serialize or deserialize the arguments of the method. Besides the normal method table, the Manta compiler generates two additional tables for objects that extend *java.rmi.Remote*: one containing pointers to the generated marshallers, and one with pointers to the unmarshallers. The additional tables are used to dispatch to the right marshaller or unmarshaller, depending on the dynamic type of the given object, thus avoiding reflection.

To detect duplicate objects, the marshalling code uses a hash table containing objects that have already been serialized. An optimization that Manta implements is the following: if the method does not contain any parameters that are objects, the cycle-check table is not built up, which makes simple methods faster. Just like with Sun RMI, the table itself is not transferred over the network; instead, it is rebuilt on-the-fly by the receiver. This is possible because objects are serialized and deserialized in the same order.

Another optimization concerns the type descriptors for the parameters of an RMI. When a serialized object is sent over the network, a descriptor of its type must also be sent. The Sun RMI protocol sends a complete type descriptor for every class used in the remote method, including the name and package of the class, a version number, and a description

of the fields in this class. All this information is sent for every RMI; information about a class is only reused within a single RMI. With the Manta RMI protocol, each machine sends the type descriptor only once to any other machine. The first time a type is sent to a certain machine, the type is given a new type-id that is specific to the sender/receiver pair, and both the type descriptor and the type-id are sent. When more objects of this type are sent to the same destination machine, only the type-id is used instead of the complete type descriptor. When the destination machine receives a type descriptor, it checks if it already knows this type. If not, it loads it from the local disk or an HTTP server. Next, it inserts the type-id and a pointer to the meta class (a data structure that contains information about a class) in a table, for future references. This scheme thus ensures that type information is sent only once to each remote node. Also, the connections require hardly any state information. The underlying Panda layer does not use TCP sockets but LFC or UDP (so it does not keep any file descriptors open). The mechanism therefore scales well.

The Manta RMI protocol avoids thread-switching overhead at the receiving node. In the general case, an invocation is serviced at the receiving node by a newly allocated thread, which runs concurrently with the application threads. With this approach, however, the allocation of the new thread and the context switch to this thread will be on the critical path of the RMI. To reduce the allocation overhead, the Manta runtime system maintains a pool of pre-allocated threads, so the thread can be taken from this pool instead of being allocated. In addition, Manta avoids the context-switching overhead for simple cases. The Manta compiler determines whether a remote method may block (whether it may execute a "wait()" operation) or run indefinitely. If the compiler can guarantee that a given method will never block or run for a long time, the receiver executes the method without doing a context switch to a separate thread. In this case, the current application thread will service the request and then continue. The compiler currently makes a conservative estimation and only guarantees the nonblocking property for methods that do not call other methods and do not create objects (since that might invoke the garbage collector, which may cause the method to block). Moreover, the compiler guarantees that the method will not run for a long time, by checking whether sure the method does not contain loops. This analysis has to be conservative, because a deadlock situation might occur if an application thread would service a method that blocks. We found that this optimization especially helps for get/set methods (methods that just read or update a field of a class), which are common in object-oriented programs. Sun RMI always starts a separate thread for each RMI.

Another optimization we performed concerns garbage collection. The problem is that the RMI mechanism can create several objects that become garbage at the end of the invocation. This happens because object parameters to RMIs are deserialized into new objects. If the object parameters are not assigned to fields that remain alive after the RMI, the objects become garbage. The Manta compiler provides support for the server side: it determines whether the objects that are argument or result of the invocation may *escape* (i.e., whether a reference to the object is retained outside the RMI invocation). If not, Manta allows such objects to be immediately returned to the heap (the objects could be allocated on the stack, but Manta currently does not support this). This substantially improved latency and throughput for RMIs that use objects as arguments or return value (see Section 3.5.4), as garbage collection overhead is reduced substantially.

```
1  import java.rmi.*;
2  import java.rmi.server.UnicastRemoteObject;
3
4  public interface ExampleIntr extends Remote {
5      public int square(int i, String p, String q)
6        throws RemoteException;
7  }
8
9  public class Example
10    extends UnicastRemoteObject implements ExampleIntr {
11     int value;
12     String name;
13
14     public int square(int i, String p, String q)
15       throws RemoteException {
16        value = i;
17        name = p + q;
18        System.out.println("i = " + i);
19        return i*i;
20     }
21  }
```

Figure 3.8: An example remote class.

The Manta RMI protocol cooperates with the garbage collector to keep track of references across machine boundaries. Manta uses a local garbage collector based on a mark-and-sweep algorithm. Each machine runs this local collector, using a dedicated thread that is activated by the runtime system or the user. The distributed garbage collector is implemented on top of the local collectors, using a reference-counting mechanism for remote objects (distributed cycles remain undetected). If a Manta process communicates with a JVM, it uses the distributed garbage collection algorithm of the Sun RMI implementation, which is based on leasing [157].

Generated Marshalling Code

Consider the *Example* class in Figure 3.8. The *square()* method can be called from another machine (because the method is mentioned in the *ExampleIntr* interface, which extends *java.rmi.Remote*), so the Manta compiler generates marshalling and unmarshalling code for it. The generated marshaller for the *square()* method is shown in Figure 3.9 in pseudo code. The *writeHeader()*, *writeInt()*, etc. calls are macros that simply save opcodes and stream information into the opcode buffer (shown in Figure 3.6). Because *square()* has *String*s as parameters (which are objects in Java), a table (named *objectTable* in the example) is built to detect duplicates (line 5). A special flag, called *CREATE_THREAD*, is set in the header data structure because *square()* potentially blocks: it contains a method call that may block (e.g., in a *wait()*), and it creates objects, which may trigger garbage collection and thus also may block. The *writeObject()* calls serialize the string objects. *flushMessage()* does the actual writing out to the network buffer, while the function *receiveMessage()* initiates reading the reply.

```
 1  int marshall_square(class_RemoteExample *this, int i,
 2                  class_java_lang_String *p,
 3                  class_java_lang_String *q) {
 4      MarshallStruct *m = allocMarshallStruct();
 5      m->objectTable = createobjectTable();
 6
 7      writeHeader(m, this, OPCODE_CALL, CREATE_THREAD);
 8      writeInt(m, i);
 9      writeObject(m, p);
10      writeObject(m, q);
11
12      /* Request message is created, now write it to the network. */
13      flushMessage(m);
14
15      receiveMessage(m);  /* Receive reply. */
16      opcode = readInt(m);
17      if (opcode == OPCODE_EXCEPTION) {
18          class_java_lang_Exception *exception;
19          exception = readObject(m);
20          freeMarshallStruct(m);
21          THROW_EXCEPTION(exception);
22      } else if (opcode == RESULT_CALL) {
23          result = readInt(m);
24          freeMarshallStruct(m);
25          RETURN(result);
26      }
27  }
```

Figure 3.9: The generated marshaller (pseudo code) for the *square* method.

Pseudo code for the generated unmarshaller is shown in Figure 3.10. The header is already unpacked when this unmarshaller is called. Because the *CREATE_THREAD* flag in the header was set, this unmarshaller will run in a separate thread, obtained from a thread pool. The marshaller itself does not know about this. Note that the *this* parameter is already unpacked and is a valid reference for the machine on which the unmarshaller will run.

Transferring the Data

When serialization and marshalling is finished, Panda uses the I/O-vector to directly copy the object data to the network interface. For an implementation on Fast Ethernet, using UDP, this requires an extra copy to a buffer, which can than be handed off to the kernel. On Myrinet, however, a zero-copy implementation is possible, and the processor on the network card can directly send the information from the object data to the network. A diagram demonstrating how this works is shown in Figure 3.11. We will explain this process in more detail below.

Panda sends the I/O-vector to the LANai processor on the Myrinet board. Next, the LANai can copy the object data from the Java heap to the packets which are located in the memory of the Myrinet board, using either DMA (Direct Memory Access) or PIO

```
1  void unmarshall_square(class_RemoteExample *this, MarshallStruct *m) {
2      m->objectTable = createobjectTable();
3
4      int i = readInt(m);
5      class_java_lang_String *p = readObject(m);
6      class_java_lang_String *q = readObject(m);
7
8      result = CALL_JAVA_FUNCTION(square, this, i, p, q, &exception);
9      if (exception) {
10        writeInt(m, OPCODE_EXCEPTION);
11        writeObject(m, exception);
12     } else {
13        writeInt(m, OPCODE_RESULT_CALL);
14        writeInt(m, result);
15     }
16
17     /* Reply message is created, now write it to the network. */
18     flushMessage(m);
19 }
```

Figure 3.10: The generated unmarshaller (pseudo code) for the *square* method.
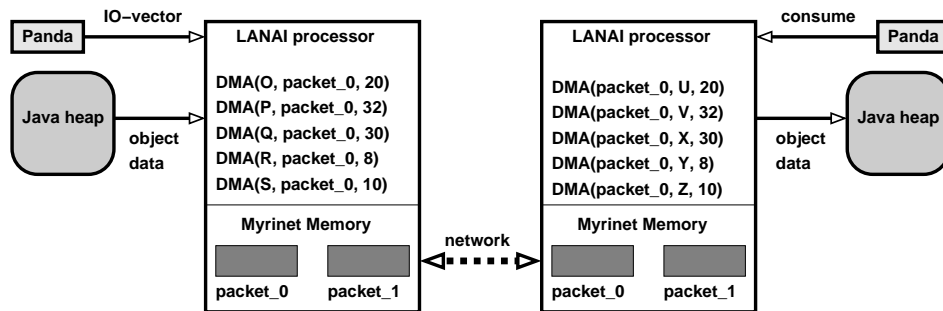
Figure 3.11: Zero-copy transfer of *Foo* with Myrinet.

(Programmed I/O). The host CPU is not involved in this copying process, and can meanwhile do useful work. When the data has been copied into a packet, the LANai sends the message over the network. The data is received in a packet in the memory of the Myrinet board on the remote host. The generated deserializer uses Panda's *consume* operation to read the data (see line 14 of Figure 3.5). Panda transfers the requested data from the network packet directly into the destination object in the Java heap.

3.3.4   Summary of Manta RMI Optimizations

As described, Manta RMI implements many optimizations over Sun RMI. We will now give a summary of these optimizations, again split into three categories: low-level communication, serialization and the RMI protocol.

Low-level communication:

- the use of an efficient user-level communication substrate;

- a scatter/gather interface for zero-copy communication.

Serialization:

- the use of compiler-generated serialization code which avoids runtime type inspection;

- avoiding the streaming layers of the JDK;

- no element-wise traversal of primitive arrays;

- no byte swapping between machines with the same byte ordering;

- during deserialization, avoid initialization of object fields and array elements (to zero or *null*) for newly created objects;

- optimization for *final* classes that eliminates virtual function calls;

- dispatch to the generated serializer of the actual dynamic type of an object, thus avoiding function calls (and I/O-vector entries) to serialize super classes.

RMI:

- compiler generated marshalling code that calls the generated serialization code to (un)marshall the RMI parameters;

- cache type descriptors, also between multiple RMIs;

- avoid thread creation by using a thread pool and by avoiding the use of threads in simple cases altogether;

- avoid creation of cycle check tables for RMIs without object parameters;

- use escape analysis to (de)allocate object parameters to RMIs, thus reducing garbage collection overhead;

- avoid the use of the Java Native Interface (JNI);

- the use of user-level thread package and locks.

3.4   Sun Compiled: Manta's Optimized Sun RMI

We will now prepare a comparison for performance analysis of the Manta RMI optimizations. A difference between Manta and the Sun JDK is that Manta uses a native compiler, whereas the JDK uses a JIT. The sequential speed of the code generated by the Manta compiler is much better than that of the Sun JIT (version 1.2), and comparable to the IBM JIT (version 1.1.8). The overhead of the Java Native Interface and differences in

sequential code speed would obscure the comparison between the Manta and Sun RMI
protocols. To allow a fair comparison and a detailed performance analysis, we therefore
built a system, which uses the Sun RMI code, compiled by the native Manta compiler.
Native interfacing in Manta comes at virtually no cost, and some of the performance
problems in the JDK implementation of Sun RMI can be resolved. This optimized system
is called *Sun compiled*. Below, we discuss the *Sun compiled* system and the optimizations
we implemented for the Sun RMI protocol on Myrinet.

3.4.1   Sun Serialization

We built a Sun RMI system by compiling the Sun RMI source code with Manta. Sun's
native code had to be replaced by new Manta native C code. This native code is used for
object serialization and interfacing to the network layer. The new object serialization code
is similar to the Sun implementation, using runtime type inspection to convert objects to
bytes. To improve performance, we re-implemented a larger part of the serialization code
in C. In the new native code, we can exploit knowledge about the memory layout of
objects: we directly use the class information and data fields present in the object, instead
of calling the reflection mechanism in Java, as the Sun JDK does.

In Java, it is impossible to create objects without initializing them first. Because
Manta implements Sun serialization natively, however, it is possible to implement the
optimization that avoids wiping objects (initializing them to zero or *null* during deseri-
alization. This optimization is also effective for Sun RMI, and increases the throughput
considerably (see Section 3.5.4).

Moreover, in the Sun JDK, expensive Java Native Interface calls are required to con-
vert the primitive types *float* and *double* to bytes before writing them to the output stream.
In the new code, these values can directly be written into the stream buffer, converting
them on-the-fly. Regardless of these optimizations, this serialization protocol is the same
as the one used in Sun JDK.

3.4.2   Interfacing to Myrinet

The JDK systems use the Java Native Interface to access the socket interface. To run
the Sun JDK, IBM JDK, and the *Sun compiled* system over Myrinet, we use a socket
interface on top of Panda/Myrinet. The socket interface is called PandaSockets and is a
re-implementation of Berkeley FastSockets [143] on top of Panda. Its main virtues are
zero-copy streams, a one-to-one mapping from socket messages to Panda messages, and
a performance quite close to that of Panda itself. The Sun API currently does not allow
replacement of its sockets with an alternative implementation like FastSockets (although
the API was clearly designed to support this), so a marginal change to the API was nec-
essary. It was sufficient to declare the constructor of class *java.net.InetAddress* as *public*.
This API problem was registered in Sun's bug database, and has been solved in recent
JDK releases.

Although the PandaSockets implementation provides zero-copy streams, the JDK
streaming layers do make copies of the data. However, the *Sun compiled* system does
not make additional copies besides those already performed in the streaming layers.

3.4.3   Optimizing the Sun Compiled System

We addressed several performance problems of Sun RMI, to allow a more fair comparison with Manta. In general, we tried to eliminate the most important sources of overhead for Sun RMI, as far as these optimizations could be done in the PandaSockets layer. In particular, we optimized the interaction with the thread package, avoiding unnecessary thread switches. Because the Sun RMI code is compiled with Manta and is linked with the Manta runtime system, the *Sun compiled* system uses more efficient user-level threads and locking than the Sun and IBM JITs, and a much faster native interface.

  We modified Sun's skeleton compiler *rmic*, so the generated skeletons use the escape analysis information that the Manta compiler provides. When no reference to object parameters is retained outside the RMI invocation, the objects are immediately returned to the heap. The result of this optimization is an improved latency and throughput for *Sun compiled* RMIs that use objects as arguments or return value (see Section 3.5.4). There is no mechanism in the JDK to explicitly free objects, so we could not apply this technique for the Sun or IBM JDK. The garbage collector is at least partially responsible for the bad performance achieved even by the fast IBM JDK system. The RMI performance of the JDKs would improve considerably when the escape analysis performed by the JITs will become powerful enough to discover whether objects that are passed as a parameter to an RMI can be immediately returned.

3.4.4   Summary of Sun Compiled Optimizations

We implemented several optimizations in the *Sun compiled* system. The optimizations are not present in Sun RMI. We will now give a summary of these optimizations.

Low-level communication:

- the use of an efficient user-level communication substrate (PandaSockets);

- a one-to-one mapping from socket messages to Panda messages, providing zero-copy streams. However, copies are made by the higher level serialization mechanism, but these are unavoidable without altering the protocol.

Serialization:

- a native implementation of the serialization protocol;

- during deserialization, avoid initializing object fields and array elements for newly created objects;

- our implementation uses runtime type inspection, but avoids Java reflection;

- use of the efficient native interface of Manta instead of the JNI;

RMI:

- use escape analysis to (de)allocate object parameters to RMIs, thus reducing garbage collection overhead;

- the use of a user-level thread package and locks.

System	Version	Network	Latency (μs)	overhead in %	Throughput (MByte/s)	% of max throughput
Sun JIT	1.2	Myrinet	1316	4145	3.8	6
IBM JIT	1.1.8		550	1674	7.9	13
Sun compiled	1.1.4		301	871	15	25
KaRMI	1.05 b		241	677	27	45
Manta RMI			37	19.3	54	90
Panda RPC	4.0		31	0	60	100
Sun JIT	1.2	Fast Ethernet	1500	767	3.1	28
IBM JIT	1.1.8		720	316	6.0	54
Sun compiled	1.1.4		515	198	7.2	65
KaRMI	1.05 b		445	157	9.8	88
Manta RMI			207	19.6	10.5	95
Panda RPC	4.0		173	0	11.1	100

Table 3.1: Null-RMI latency and throughput on Myrinet and Fast Ethernet.

3.5   Performance Analysis

In this section, the communication performance of Manta RMI is compared against several RMI implementations. For the comparison, we used three systems that use Sun RMI: the Sun (Blackdown) JDK 1.2 with JIT, the IBM JDK 1.1.8 also with JIT, and a Sun RMI system (based on the JDK 1.1) compiled with our native Manta compiler. For all three systems, we built an interface to run over Myrinet. We described these systems in detail in Section 3.4, including important optimizations. We also compare with Karlsruhe RMI (KaRMI [123]), an alternative (and not protocol compatible) RMI system which has implementations on both Fast Ethernet and Myrinet. The Fast Ethernet implementation is written in Java, while the Myrinet implementation contains some native code, and is built on top of GM (Myricom's standard communication software for Myrinet). We use the Sun JDK 1.2 to run KaRMI, because KaRMI does not work with the IBM JIT (version 1.1.8).

First we will compare the performance of the different RMI implementations, using some low-level benchmarks. Since the Sun system compiled with Manta turned out to be the fastest of the three RMI systems, we use this system in the following sections to represent Sun's RMI protocol. Next, we discuss the latency (Section 3.5.2) and throughput (Section 3.5.3) obtained by Manta RMI and Sun RMI in more detail. In Section 3.5.4 we analyze the impact of several optimizations in the protocols. Finally, we evaluate the impact of the optimized RMI system at the application level, using six applications in Section 3.6. All measurements were done on the DAS system (200 MHz Pentium Pro machines). The DAS is described in more detail in Section 1.9.1.

3.5.1   Performance Comparison with different RMI Systems

Table 3.1 shows the null-RMI latency (i.e., the round-trip time of an RMI without parameters or a return value) and throughput of various RMI implementations on Myrinet and Fast Ethernet. All tests first run a number iterations to "warm up" the JIT, only then is

	Manta				Sun Compiled			
	empty	1 obj	2 obj	3 obj	empty	1 obj	2 obj	3 obj
Serialization overhead	0	6	10	13	0	195	210	225
RMI Overhead	5	10	10	10	180	182	182	184
Communication overhead	32	34	34	35	121	122	124	125
Method call	0	1	1	1	0	1	1	1
Total	37	51	55	59	301	500	517	535

Table 3.2: Breakdown of Manta and *Sun Compiled* RMI on Myrinet (times are in μs).

the real measurement started. Furthermore, the tests are performed many times to factor out the influence of the garbage collector. On Myrinet, Manta RMI obtains a null-RMI latency of 37 μs, while the Sun JDK 1.2 (with just-in-time compilation enabled) obtains a latency of 1316 μs, which is 35 times higher. *Sun compiled* obtains a null-RMI latency of 301 μs, which is still 8 times slower than Manta RMI. In comparison, the IBM JIT 1.1.8 obtains a latency of 550 μs. The *Sun compiled* system uses more efficient locks than the Sun and IBM JITs (using Manta's user space thread package) and a much faster native interface. Also, it reduces the garbage collection overhead for objects passed to RMI calls. Finally, the performance of its generated code is much better than that of the Sun JDK JIT and comparable to that of the IBM JIT. The table also gives the performance of a conventional Panda Remote Procedure Call protocol, which is implemented in C. As can be seen, the performance of the Manta protocol comes close to that of Panda RPC, which is the upper bound of the performance that can be achieved with Manta RMI.

The throughput obtained by Manta RMI (for sending a large array of integers) is also much better than that of the Sun JDK: 54 MByte/s versus 3.8 MByte/s (over Myrinet). The throughput of *Sun compiled* is 15 MByte/s, 3.6 times less than for Manta RMI, but better than the Sun JDK. The table also gives performance results on Fast Ethernet. Here, the relative differences are smaller, because the low-level communication costs are higher. As the *Sun compiled* system is by far the most efficient implementation of the Sun RMI protocol, we use this system in the following sections to represent Sun's protocol. This approach is thus extremely favorable for Sun RMI, and in reality, the performance difference between Sun RMI and Manta RMI is even larger.

KaRMI uses a different protocol than Sun RMI, and achieves a performance that is slightly better than *Sun Compiled*. However, on Myrinet, the latency of KaRMI is 6.5 times higher than the Manta RMI latency, while the throughput achieved by KaRMI is only half that of Manta RMI. Moreover, the numbers in the table show the KaRMI numbers for only 1000 calls, because KaRMI gets slower when more calls are done (e.g., when 100000 RMIs are done, the latency is increased with 28 μs). We suspect that KaRMI does not deallocate some resources. On Fast Ethernet, the KaRMI latency is also more than two times higher than the Manta RMI latency. In fact, the Manta RMI latency on Fast Ethernet is already lower that the KaRMI latency on Myrinet.

3.5.2   Latency

We first present a breakdown of the time that Manta and *Sun compiled* spend in remote method invocations. We use a benchmark that has zero to three empty objects (i.e., objects with no data fields) as parameters, while having no return value. The benchmarks are written in such a way that they do not trigger garbage collection. The results are shown in Table 3.2. The measurements were done by inserting timing calls, using the Pentium Pro performance counters, which have a granularity of 5 nanoseconds on our hardware. The *serialization overhead* includes the costs to serialize the arguments at the client side and deserialize them at the server side. The *RMI overhead* includes the time to initiate the RMI call at the client, handle the upcall at the server, and process the reply (at the client), but excludes the time for (de)serialization and method invocation. The *communication overhead* is the time from initiating the I/O transfer until receiving the reply, minus the time spent at the server side. For Manta, the measurements do not include the costs for sending type descriptors (as these are sent only once).

The simplest case is an empty method without any parameters, the null-RMI. On Myrinet, a null-RMI takes about 37 μs with Manta. Only 6 μs are added to the round-trip latency of the Panda RPC, which is 31 μs. Manta RMI sends slightly more data than a Panda RPC (e.g., the *CREATE_THREAD* flag, etc.), hence the communication takes 32 μs instead of 31 μs.

The relatively large difference between passing zero or one object parameters can be explained as follows. Separate measurements (not shown) indicate that almost all time that is taken by adding object parameters is spent at the remote side of the call, deserializing the call request. The serialization of the request on the calling side is affected less by the object parameters, for several reasons. First, the runtime system has to build a table to detect possible cycles and duplicates in the objects. Second, RMIs containing object parameters are serviced by a dedicated thread from a thread pool, because such RMIs may block by triggering garbage collection. The thread-switching overhead in that case is about 5 μs. Finally, the creation of the parameter object also increases the latency.

For *Sun compiled*, a null-RMI over Myrinet takes 301 μs, which is 8 times slower than Manta, even with all the optimizations we applied. Manta RMI obtains major performance improvements in all layers: compiler-generated serializers win by a factor of 17 or more; the RMI overhead is 18 times lower; and the communication protocols are 4 times faster. The table also shows how expensive Sun's serialization and RMI protocol (stream and dispatch overhead) are, compared to Manta. With three object parameters, for example, the total difference is a factor 17.8 (409 μs versus 23 μs).

3.5.3   Throughput

Next, we study the RMI throughput of Manta, *Sun compiled* and KaRMI. We use a benchmark that measures the throughput for a remote method invocation with various types of arguments and no return value. Because RMIs are synchronous, however, the sender does wait for the remote method to return. The benchmark performs 10,000 RMIs, with 100 KBytes of arguments each, while the return type is *void*. In Manta, all arrays of primitive types are serialized without conversion or byte swapping, so the actual type is irrelevant.

	Manta		Sun *Compiled*		KaRMI	
	throughput	% of max	throughput	% of max	throughput	% of max
bytes[]	54	90	37	62	27	45
int[]	54	90	15	25	8.0	13
doubles[]	54	90	15	25	2.8	4
tree payload	2.7	34	0.6	8	0.1	1
tree total	4.1	52	0.9	11	–	–

Table 3.3: RMI throughput (in MByte/s) on Myrinet of Manta, *Sun Compiled* and KaRMI for different parameters.

The results are shown in Table 3.3. Throughput results for KaRMI were measured with the Sun 1.2 JIT. Manta achieves a throughput of 54 MByte/s for arrays of integers, compared to 60 MByte/s for the underlying Panda RPC protocol (see Table 3.1). In comparison, the throughput of *Sun compiled* is only 15 MByte/s, while KaRMI's throughput is even less (8.0 MByte/s).

The throughput for *Sun compiled* for arrays of integers is substantially higher than for the Sun JIT (15 MByte/s versus 3.8 MByte/s, see Table 3.1), due to our optimizations described in Section 3.4. Still, the throughput for *Sun compiled* is much lower than that for Manta RMI. The Sun serialization protocol internally buffers messages and sends large messages in chunks of 1KB, which decreases throughput. Even more important, Sun RMI (and *Sun compiled*) performs unnecessary byte swapping. The sender and the receiver use the same format for integers, but this format differs from the standard RMI format. Then, *Sun compiled* uses serialization to convert the data to the standard format. Manta RMI, on the other hand, always sends the data in the format of the sender, and lets the receiver do byte swapping only when necessary. The throughput obtained by the *Sun compiled* system for an array of bytes, for which no byte swapping is needed, is 37 MByte/s (see Table 3.3). This throughput is high because all I/O layers in the Sun system have short-cuts for byte arrays. When writing a byte array, each I/O layer passes the buffer directly to the layer below it, without copying. Similarly, when a read request for a byte array is done, it is passed on to the bottom layer, and the result is passed back up, without copying.

The throughput numbers for KaRMI are much worse than the numbers for Manta RMI and *Sun compiled*. These results can partly be caused by the bad Sun JIT performance. In [130], the low throughput is attributed to the overhead of thread scheduling and the interaction between Java threads and system threads. In Chapter 7, we will evaluate the performance of KaRMI with the latest JITs and hardware.

The binary tree throughput benchmark is based on the KaRMI latency benchmark described in [123], but using input parameters and no return values. The benchmark sends balanced trees with 1023 nodes, each containing four integers. The reported throughput is that for the user "payload" (i.e., the four integers), although more information is sent over the network to rebuild the tree structure. When known, we also show the throughput of the total data that is sent over the network (including protocol data and headers). We also implemented the tree test using Panda directly. The Panda test receives the data in

	Manta RMI			Sun Compiled RMI		
	empty	int[100]	1 obj	empty	int[100]	1 obj
Bytes (using type descriptor)	44	484	96	63	487	102
Bytes (using type-id)	44	452	64	-	-	-
Writing type descriptor (μs)	-	11	12	-	25	27
Reading type descriptor (μs)	-	15	17	-	55	73

Table 3.4: Amount of data sent by Manta RMI and Sun RMI and runtime overhead of type descriptors.

a pre-allocated tree, and is an upperbound of the performance that can be achieved with Manta. The Panda throughput for binary trees is 7.9 MBytes/s.

For KaRMI, we use a tree of only 63 nodes, because larger trees do not work. We suspect that KaRMI does not implement fragmentation, because, when using GM, it exits with a "message to large" error. On TCP, the test does work with 1023 nodes, and achieves exactly the same performance (i.e., 0.1 MByte/s), as the Myrinet test with 63 nodes. This is the expected behavior, because the network is not the limiting factor in this test.

The throughput for this benchmark is very low in comparison with the throughput achieved for arrays. The overhead can be attributed to the small size of the nodes and the dynamic nature of this data type, which makes especially (de)serialization expensive: the tree is written to and read from the network buffer a tree node at a time, and for Sun serialization (and thus *Sun compiled*) even a byte at a time; therefore the overhead of network access is incurred much more often than for arrays. Still, the Panda throughput for trees (7.9 MByte/s), is significantly higher than the Manta RMI throughput: Manta achieves only 52 % of the Panda throughput. This is caused by the fact that Java serialization allocates new tree nodes during deserialization, and also has to garbage collect the tree nodes that are no longer used, while the Panda test receives the tree data in a pre-allocated data structure.

The zero-copy implementation adds an entry to a Panda I/O-vector for each object (shown in Figure 3.6), which contains only 16 bytes (the 4 integers). For such small data sizes, the I/O-vectors have a relatively large overhead. Therefore, for Manta RMI, we also experimented with making an extra copy in a buffer for objects. With this approach, a single I/O-vector entry is added for the entire buffer containing all object fields. However, we found that the break-even point lies at a data size of exactly 16 bytes. For smaller data elements, making a copy is more efficient, while for larger data items using I/O-vectors is faster. For 16 byte data elements, as used in our benchmark, the performance of both approaches is identical.

3.5.4   Impact of Specific Performance Optimizations

The Manta protocol performs various optimizations to improve the latency and throughput of RMIs. Below, we analyze the impact of specific optimizations in more detail.

Type Descriptors

As explained in Section 3.3.3, the Sun protocol always sends a complete type descriptor for each class used in the RMI. Manta RMI sends this type information only once for each class; it uses a type-id in subsequent calls. The amount of data that Manta RMI sends for object and array parameters thus depends on whether a parameter of the same class has been transmitted before. Table 3.4 shows the amount of data sent for both cases, for both Manta RMI and *Sun compiled* RMI. For each case, the table gives the number of bytes for RMIs with no arguments, with a 100-element array of integers, and with a single object containing an integer and a double. It also shows the times on a 200MHz Pentium Pro to write the type descriptor at the sending side and to read it at the receiving side.

    As can be seen, the type-descriptor optimization saves 32 bytes for each array or object parameter. The run time costs saved by the optimization for reading and writing the type descriptors is 26 μs for arrays (11 + 15) and 29 μs for objects (12 + 17). Moreover, a type descriptor includes the name of its class. We used a single-letter class name (and no package) in the benchmark, so Table 3.4 shows the best possible numbers for Sun RMI, the optimization in the Manta RMI protocol gains even more for classes with longer names. For example, consider the frequent case of sending a String parameter. The class name that will be sent over the network then is *java.lang.String*, resulting in 15 additional bytes to be sent, leading to a total of 117 bytes. Manta, however, still sends only 64 bytes when a type-descriptor is used.

    The Sun RMI protocol sends more data than the Manta protocol, but, more important, it spends a considerable amount of time in processing and communicating the data. Package and class names for instance, are transferred in the UTF-8 format (a transformation format of Unicode and ISO 10646). Converting strings to UTF-8 and back requires some processing time. The Sun protocol spends 80 μs handling the type descriptors for arrays and 100 μs for objects. It pays this price at every invocation, whereas the Manta protocol incurs the (already lower) overhead only once.

Using a Scatter/Gather Interface

As explained in Section 3.3.1, the Panda library on which Manta is built uses a scatter/gather interface to minimize the number of memory copies needed. This optimization increases the throughput for Manta RMI. To assess the impact of this optimization we also measured the throughput obtained when the sender makes an extra memory copy. In this case, the maximum throughput decreases from 54 to 44 MByte/s, because memory copies are expensive on a Pentium Pro [28]. Another memory copy on the receiving side (which is also avoided by the scatter/gather interface of Panda) has a similar impact. This experiment thus clearly shows the importance of the scatter/gather interface. Unfortunately, dereferencing the I/O-vector involves extra processing, so the null-RMI latency of the current Manta RMI system is slightly higher than that for an earlier Panda version without the scatter/gather interface (34 versus 37 μs) [113].

	Manta RMI		*Sun compiled*	
	No escape analysis	With escape analysis	No escape analysis	With escape analysis
Bytes	29	54	20	38
Ints	29	54	14	15
Doubles	29	54	14	15

Table 3.5: Throughput (in MByte/s) on Myrinet of Manta RMI and Sun RMI (compiled) for different parameters.

Reducing Byte Swapping

Another optimization that increases the throughput is to avoid byte swapping between identical machines. On our hardware, Sun RMI does two byte swapping operations for each primitive value that is transferred, while Manta RMI does none (see Section 3.3.2).

We measured the impact of this optimization by adding byte-swapping code to the sender side of Manta RMI. (This code is not present in the normal Manta system, since the sender never does byte swapping with Manta.) Also, an additional copy must be made on the sending side, because the source array must not be changed. If byte swapping is performed by both the sender and receiver (as Sun RMI does), the throughput of Manta RMI for arrays of integers or floats decreases by almost a factor two. The maximum throughput obtained with byte swapping enabled is decreased from 54 to 30 MByte/s. This experiment clearly shows that unnecessary byte swapping adds a large overhead, which is partly due to the extra memory copies needed.

Avoiding the initialization of Objects

Another important optimization we implemented is to avoid the initialization of objects and arrays during deserialization (see Section 3.3.2). We found that this optimization has a large impact on the throughput. For arrays, the throughput drops from 54 to 41 MByte/sec when the arrays are initialized first.

Escape Analysis

As described in Section 3.4, we implemented a simple form of escape analysis. With this analysis, objects that are argument or result of an RMI but that do not escape from the method will be immediately returned to the heap. Without this optimization, such objects would be subject to garbage collection, which reduces the RMI throughput. Table 3.5 shows the impact of this optimization. Without escape analysis the throughput for Manta is reduced from 54 to 29 MByte/s. For *Sun compiled*, the throughput for byte arrays is reduced from 38 to 20 MByte/s, but the other throughput numbers are hardly affected (because these cases also suffer from other forms of overhead, in particular byte swapping).

application	problem size
SOR	a 578×578 grid
ASP	a 1280-node graph
RADIX	an array with 3,000,000 numbers
FFT	a matrix with 2^{20} elements
WATER	1728 bodies
BARNES	30,000 bodies

Table 3.6: Problem sizes for the applications in this chapter.

3.6   Application Performance

The low-level benchmarks show that Manta obtains a substantially better latency and throughput than the Sun RMI protocol. For parallel programming, however, a more relevant metric is the efficiency obtained with applications. To determine the impact of the RMI protocol on application performance, we have written six parallel applications with different granularities. The applications are described in Section 2.7.

We briefly discuss the performance of the applications using Manta and *Sun compiled*. Each application program typically first creates the remote objects needed for inter-process communication and exchanges the references to these objects among the machines. Therefore, the overhead of distributed garbage collection and reference counting only occurs during initialization and has hardly any impact on application performance. The problem sizes we used are shown in Table 3.6.

Table 3.7 shows the run times, while Figure 3.12 shows the speedups for these six applications obtained by Manta and *Sun compiled*. For both systems, the programs are compiled statically using the Manta compiler. The speedups for each system are computed relative to the parallel Manta RMI program on a single machine. The sequential execution times of Manta RMI and *Sun compiled* are very similar, as the applications are compiled with the Manta compiler for both systems (for some applications Manta RMI is slightly faster due to caching effects).

Table 3.7 also gives communication statistics for the six applications, including the total number of messages sent (summed over all machines) and the amount of data transferred, using 16 or 32 machines. These numbers were measured at the Panda layer, so they include all RMI header and RMI protocol data. Also, a Manta RMI generates two Panda messages: a request and a reply.

Figure 3.12 and Table 3.7 show that Manta RMI's higher communication performance results in substantially better application performance. *Sun compiled* performs reasonably well for only two applications: Water and ASP. Water has by far the lowest communication overhead of the six applications. On 32 machines, it sends 2708 (44984/16.61) messages and 1.20 (20.05/16.61) megabytes per second (for *Sun compiled*). ASP communicates more than Water, but it performs relatively few RMIs per second. With the other four applications, Manta RMI obtains much better performance. Barnes-Hut and Radix are 2.4 faster with Manta RMI, SOR is even 4.3 times faster on 32 machines.

Manta RMI obtains high efficiencies for all applications except Radix sort. Radix sends the largest number and volume of messages per second of all six applications; on 32
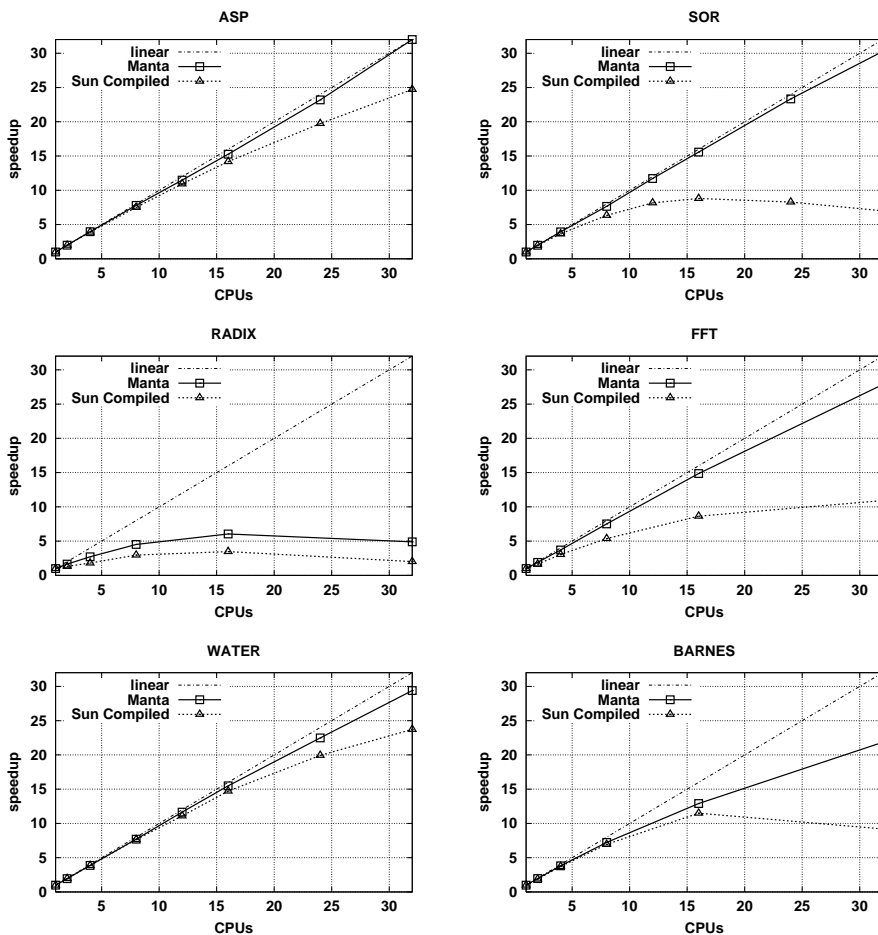
Figure 3.12: Speedups of 6 RMI applications with Manta RMI and *Sun compiled*.

machines, it sends almost 27,000 (39418/1.46) messages and 85 (124.68/1.46) megabytes per second, summed over all machines.

Table 3.7 shows that the Sun RMI protocol sends far more messages for all applications than Manta RMI. The reason is that the Sun serialization protocol buffers messages and transfers large messages in chunks of 1 KB (see Section 3.5.3). The volume of the data transferred by the Manta protocol is somewhat lower than that for the Sun protocol, because Manta RMI does not send type descriptors for each class on every call, and because Manta RMI sends fewer messages and thus fewer headers.

| Program | System | 16 machines | | | 32 machines | | |
		Time (s.)	# Messages	Data (MByte)	Time (s.)	# Messages	Data (MByte)
ASP	Manta	25.64	38445	95.30	12.22	79453	196.96
	Sun compiled	27.56	154248	100.23	15.83	319870	207.17
SOR	Manta	10.96	44585	80.38	5.64	92139	166.11
	Sun compiled	19.38	134765	84.62	24.39	285409	175.11
Radix	Manta	1.19	9738	62.46	1.46	39418	124.68
	Sun compiled	2.06	78674	64.87	3.56	183954	130.35
FFT	Manta	3.52	8344	152.06	1.88	33080	157.14
	Sun compiled	6.07	173962	157.37	4.80	204949	163.45
Water	Manta	25.46	6023	8.44	13.41	23319	17.30
	Sun compiled	26.78	16088	9.59	16.61	44984	20.05
Barnes	Manta	14.90	18595	23.78	8.81	107170	52.26
	Sun compiled	16.74	45439	25.20	20.91	171748	57.58

Table 3.7: Performance data for Manta and *Sun Compiled* on 16 and 32 machines

3.7   Related Work

We discuss related work in two areas: optimizations for RMI, and fast communication systems.

3.7.1   Optimizations for RMI

RMI performance is studied in several other papers. KaRMI is a new RMI and serialization package (drop-in replacement) designed to improve RMI performance [123, 130]. The performance of Manta is better than that of KaRMI (see Table 3.3 in Section 3.5.3). The main reasons are that Manta uses static compilation and a completely native runtime system (implemented in C). Also, Manta exploits features of the underlying communication layer (the scatter/gather interface). KaRMI uses a runtime system written mostly in Java (which thus suffers from the poor JIT performance). Manta, on the other hand, was developed from scratch to obtain high communication performance, and implements many optimizations to make the frequent case fast. Both KaRMI and Manta RMI use a wire format that is different from the standard RMI format.

Krishnaswamy et al. [93] improve RMI performance somewhat by using caching and UDP instead of TCP. Their RMI implementation, however, still has high latencies (e.g., they report null-RMI latencies above a millisecond on Fast Ethernet). Also, the implementation requires some modifications and extensions of the interfaces of the RMI framework. Javanaise [71] and VJava [106] are other Java systems that implement object caching. Javanaise proposes a new model of distributed shared objects (as an alternative to RMI). Sampemane et al. [148] describe how RMI can be run over Myrinet using the *socket-Factory* facility. Breg et al. [27] study RMI performance and inter operability. Hirano et al. [75] provide performance figures of RMI and RMI-like systems on Fast Ethernet. Manta differs from the above systems by being designed from scratch to provide high performance, both at the compiler and runtime system level.

3.7.2   Fast Communication Systems

Much research has been done since the 1980's on improving the performance of Remote Procedure Call protocols [77, 80, 140, 151, 162]. Several important ideas resulted from this research, including the use of compiler-generated (un)marshalling routines, avoiding thread-switching and layering overhead, and the need for efficient low-level communication mechanisms. Many of these ideas are used in today's communication protocols, including RMI implementations.

Except for the support for polymorphism, Manta's compiler-generated serialization is similar to Orca's serialization [10]. The optimization for nonblocking methods is similar to the single-threaded upcall model [97]. Small, nonblocking procedures are run in the interrupt handler to avoid expensive thread switches. Optimistic Active Messages is a related technique based on rollback at run time [168].

Instead of kernel-level TCP/IP, Manta uses Panda on top of LFC, a highly efficient user-level communication substrate. Lessons learned from the implementation of other languages for cluster computing were found to be useful. These implementations are built around user-level communication primitives, such as Active Messages [49]. Examples are Concert [86], CRL [81], Orca [12], Split-C [41], and Jade [142]. Other projects on fast communication in extensible systems are SPIN [17], Exo-kernel [84], and Scout [117].

Several projects are currently also studying protected user-level network access from Java, often using VIA [38, 39, 170]. However, these systems do not yet support Remote Method Invocation.

3.8   Conclusion

In this chapter, we investigated how to implement Java's Remote Method Invocation efficiently, with the goal of using this flexible communication mechanism for parallel programming. Reducing the overhead of RMI is more challenging than for other communication primitives, such as Remote Procedure Call, because RMI implementations must support inter operability and polymorphism. Our approach to this problem is to make the frequent case fast. We have designed a new RMI protocol that supports highly efficient communication between machines that implement our protocol. Communication with Java virtual machines (running the Sun RMI protocol) is also possible but slower. As an example, all machines in a parallel system can communicate efficiently using our protocol, but they still can communicate and inter operate with machines running other Java implementations (e.g., a visualization system). We have implemented the new RMI protocol (called Manta RMI) in a compiler-based Java system, called Manta, which was designed from scratch for high-performance parallel computing. Manta uses a native Java compiler, but to support polymorphism for RMIs with other Java implementations, it is also capable of dynamically compiling and linking bytecode.

The efficiency of Manta RMI is due to three factors: the use of compile-time type information to generate specialized serializers, a streamlined and efficient RMI protocol, and the use of fast communication protocols. To understand the performance implications of these optimizations, we compared the performance of Manta with that of the Sun RMI protocol. Unfortunately, current implementations of the Sun protocol are inefficient,

making a fair comparison a difficult task. To address this problem, we have built an implementation of Sun RMI by compiling the Sun protocol with Manta's native compiler. We also reduced the overhead of this system for native calls, thread switching, and temporary objects. This system, called *Sun compiled*, achieves better latency and throughput than the Sun JDK and the IBM JIT, so we used this system for most measurements in the paper, in particular to compare the Manta and Sun RMI protocols.

The performance comparison on a Myrinet-based Pentium Pro cluster shows that Manta RMI is substantially faster than the compiled Sun RMI protocol. On Myrinet, the null-RMI latency is improved by a factor of 8, from 301 μs (for *Sun compiled*) to 37 μs, only 6 μs slower than a C-based RPC. A breakdown of Manta and *Sun compiled* shows that Manta obtains major performance improvements in all three layers (the compiler-generated serializers, the RMI layer itself, and the low-level communication protocols). The differences with the original Sun JDK 1.2 implementation of RMI are even higher; for example, the null-RMI latency of the JDK over Myrinet is 1316 μs, 35 times as high as with Manta. The throughput obtained by Manta RMI also is much better than that of *Sun compiled*. In most cases, the Sun protocol performs unnecessary byte swapping, resulting in up to three times lower throughput than achieved by Manta.

Although such low-level latency and throughput benchmarks give useful insight into the performance of communication protocols, a more relevant factor for parallel programming is the impact on application performance. We therefore implemented a collection of six parallel Java programs using RMI. Performance measurements on up to 32 CPUs show that five out of these six programs obtain high efficiency (at least 75%) with Manta, while only two applications perform well with *Sun compiled*. Manta obtains up to 3.4 times higher speedups for the other four applications.

The results show that the current implementations of the Sun RMI protocol are inefficient. The Sun compiled system outperforms the RMI implementations of the Sun JDK and the IBM JIT. Moreover, by comparing Manta RMI with the Sun compiled system, we show that the Sun RMI *protocol* is inherently inefficient. It does not allow efficient implementations of RMI, because it enforces byte swapping, sends type information multiple times, and makes a zero-copy implementation impossible.

In conclusion, our work has shown that RMI, when the protocol is optimized, can be implemented almost as efficiently as Remote Procedure Call, even on high-performance networks like Myrinet, while keeping the inherent advantages of RMI (polymorphism and inter operability). These results suggest that an efficient RMI implementation is a good basis for writing high-performance parallel applications. In the next chapter, we will investigate whether RMI is also a good basis for doing parallel computing on grids.

Chapter 4

Wide-Area Parallel Programming with RMI: a Case Study

```
Men are more apt to be mistaken in their generalizations than in
their particular observations.

- Niccolo Machiavelli
```

Java's support for parallel processing and heterogeneity make it an attractive candidate for grid computing. Java offers two primitives that can be used for parallel computing: multithreading and Remote Method Invocations (RMI) [157]. Running multithreaded Java programs on a grid would require a DSM system that runs over wide-area networks, which is not (yet) feasible (see Section 2.3.4). Because TCP and UDP are too low-level for parallel programming, this implies that RMI is the only primitive in standard Java that could be useful for grid computing. Moreover, RPC in general is proposed as a building block for grid applications in [101, 102, 150].

In this chapter, we show how wide-area parallel applications can be expressed and optimized using Java RMI and we discuss the performance of several parallel Java applications on the DAS (see Section 1.9.1). All measurements presented in this chapter are thus performed in a *real* grid environment, using the physical wide-area links. We also discuss some shortcomings of the Java RMI model for wide-area parallel computing and how this may be overcome by adapting features from alternative programming models.

The contributions of this chapter are the following:

1. We demonstrate that RMI (or RPC in general) can indeed be used for programming distributed supercomputing applications.

2. We present performance results and hands-on experiences with the use of Java RMI wide-area parallel programming on a *real* grid system.

	Myrinet		ATM	
	Latency (μs)	Throughput (MByte/s)	Latency (μs)	Throughput (MByte/s)
Manta	37	54	4350	0.55
Sun JDK 1.2	1316	3.8	5570	0.55

Table 4.1: RMI round-trip latency and maximum throughput of Manta and Sun JDK.

3. We identify several shortcomings of using RMI as a grid programming model, and
   provide suggestions how these may be overcome.

The outline of the chapter is as follows. In Section 4.1 we briefly describe the performance of Java RMI on our measurement platform, the DAS. In Section 4.2 we describe our experiences in implementing four wide-area parallel applications in Java and we discuss their performance. In Section 4.3 we discuss which alternative programming models may contribute to an RMI-based programming model. We discuss related research in Section 4.4. In Section 4.5 we give our conclusions.

4.1   RMI Performance on the wide-area DAS system

The implementation of Manta RMI on the wide-area DAS system was already described in Section 1.4. Table 4.1 shows the round-trip latency and throughput obtained by Manta RMI and Sun RMI over the Myrinet LAN and the ATM WAN. The latencies are measured for null-RMIs, which take zero parameters and do not return a result. The maximum throughputs are measured for RMIs that take a large byte array as parameter.

For inter-cluster communication over ATM, we used the wide area link between the DAS clusters at VU Amsterdam and TU Delft (see Figure 1.2), which has the longest latency (and largest distance) of the DAS wide-area links. The difference in wide-area RMI latency between Manta and the JDK is 1.2 milliseconds. Both Manta and the JDK achieve a maximum wide-area throughput of 0.55 MByte/sec, which is almost 75% of the hardware bandwidth (6 Mbit/sec). The differences in wide-area latency between Manta and the JDK are due to Manta's more efficient serialization and RMI protocols, since both systems use the same communication layer (TCP/IP) over ATM.

4.2   Application Experience

We implemented several parallel Java applications that communicate via RMI. Initially, the applications were designed for homogeneous (local area) networks. We adapted four of these single-cluster versions to exploit the hierarchical structure of the wide-area system by minimizing the communication overhead over the wide area links, using optimizations similar to those described in [13, 132]. Below, we briefly discuss the original (single cluster) applications as well as the wide-area optimized programs and we give performance measurements on the DAS system. We only present results for Manta RMI, as we have shown in the previous chapter that Manta RMI is much faster than Sun RMI.
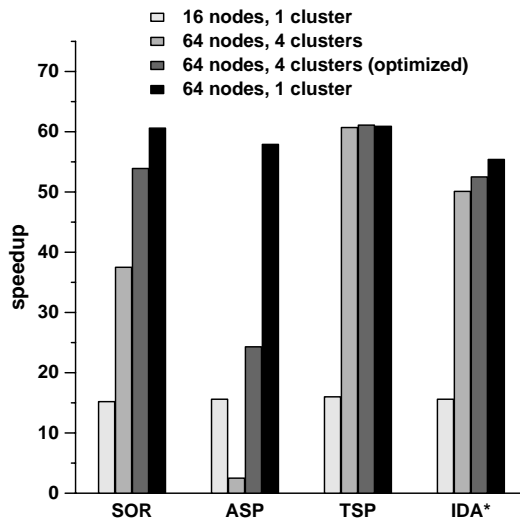
Figure 4.1: Speedups of four Java applications on a single cluster of 16 nodes, 4 WAN-connected clusters of 16 nodes each (original and optimized program), and a single cluster of 64 nodes.

For each of the four programs, we will analyze its performance on the wide-area DAS system, using the following approach. The goal of wide-area parallel programming is to obtain higher speedups on multiple clusters than on a single cluster. In fact, the speedup should be as close as possible to the speedup that could be achieved on a single large cluster with the same total number of machines. Therefore, we have measured the speedups of each program on a single DAS cluster and on four DAS clusters, the latter with and without wide-area optimizations. In addition, we have measured the speedups on a single (large) cluster with the same total number of nodes, to determine how much performance is lost by using multiple distributed clusters instead of one big local cluster. All speedups are computed relative to the same parallel program on a single machine. Due to caching effects, the run times of the different parallel versions (i.e., with and without wide-area optimizations) on one machine sometimes differ slightly.

The results are shown in Figure 4.1. The figure contains four bars for each application, giving the speedups on a single cluster of 16 nodes, four clusters of 16 nodes each (with and without wide-area aware optimizations), and a single cluster of 64 nodes. The difference between the first two bars thus indicates the performance gain or loss by using multiple 16-node clusters at different locations, instead of a single 16-node cluster, without any change to the application source. The performance gain achieved by the wide-area optimizations can be seen from the difference between the second and the third bar of each application. Comparing the second and the third bars with the fourth bar shows how much performance is lost (without and with wide-area optimizations) due to the slow wide-area network. (The 64-node cluster uses the fast Myrinet network between all nodes.)

4.2.1   Successive Over-Relaxation

Red/black Successive Over-Relaxation (SOR) is an iterative method for solving discretized Laplace equations on a grid data structure (see Section 2.7.1). Here, it is used as an example of nearest neighbor parallelization methods. SOR is an iterative algorithm that performs multiple passes over a rectangular grid, until the grid changes less than a certain value, or a fixed number of iterations has been reached. The new value of a grid point is computed using a stencil operation, which depends only on the previous value of the point itself and its four neighbors on the grid.

The skeleton code for the single-cluster parallel Java program for SOR is given in Figure 4.2. The interface *BinIntr* contains the methods *put* and *get* to mark them as RMI methods. The parallel algorithm we use distributes the grid row-wise among the available processors, so each machine is assigned several contiguous rows of the grid, denoted by the interval *LB* to *UB* (for lower bound and upper bound). Each processor runs the *run* method of class *SOR*, which performs the SOR iterations until the program converges, or until a fixed number of iterations is reached. Each iteration has two phases, for the red and black grid points. Due to the stencil operations and the row-wise distribution, every process needs one row of the grid from its left neighbor (row $LB - 1$) and one row from its right neighbor (row $UB + 1$). Exceptions are made for the first and last process, but we have omitted this from our skeleton code for brevity.

At the beginning of every iteration, each processor exchanges rows with its left and right neighbors via RMI and then updates its part of the grid using this boundary information from its neighbors. The row exchange is implemented using a remote object of class *Bin* on each processor. This object is a buffer that can contain at most one row. It has synchronized methods to put and get data.

On a local cluster with a fast switch-based interconnect (like Myrinet), the exchange between neighbors adds little communication overhead, so parallel SOR obtains high speedups. On a wide-area system, however, the communication overhead between neighbors that are located in different clusters will be high, as such communication uses the WAN. The Java program allocates neighboring processes to the same cluster as much as possible, but the first and/or last process in each cluster will have a neighbor in a remote cluster.

To hide the high latency for such inter-cluster communication, the wide-area optimized program uses split-phase communication for exchanging rows between clusters, as shown in Figure 4.3. It first initiates an asynchronous send for its boundary rows and then computes on the inner rows of the matrix. When this work is finished, a blocking receive for the boundary data from the neighboring machines is done, after which the boundary rows are computed.

The optimization is awkward to express in Java, since Java lacks asynchronous communication. It is implemented by using a separate thread (of class *SenderThread*) for sending the boundary data. How threads work in Java was explained in Chapter 2. To send a row to a process on a remote cluster, the row is first given to a newly created SenderThread; this thread will then put the row into the Bin object of the destination process on a remote cluster, using an RMI. In our implementation, instead of creating a new thread, a pre-created thread from a thread pool is used. For simplicity, this is not

```
1  public interface BinIntr extends java.rmi.Remote {
2      public void put(double [] row);
3  }
4
5  public class Bin extends UnicastRemoteObject implements BinIntr {
6      public synchronized void put(double [] row) {
7          // Wait until the bin is empty and save the new row.
8      }
9
10     public synchronized double [] get() {
11         // Wait until the bin is full and return the row.
12     }
13 }
14
15 public class SOR {
16     private BinIntr leftBin, rightBin; // Remote bins of neighbors.
17     private Bin myLeftBin, myRightBin; // My own bin objects.
18     private double[][] matrix; // The matrix we are calculating on.
19
20     public void sendRows() {
21         // synchronous RMI (first row of my partition)
22         leftBin.put(matrix[LB]);
23
24         // synchronous RMI (last row of my partition)
25         rightBin.put(matrix[UB]);
26     }
27
28     public void receiveRows() {
29         matrix[LB-1] = myLeftBin.get();
30         matrix[UB+1] = myRightBin.get();
31     }
32
33     public void run() {
34         do {     // do red/black SOR on the interval LB .. UB
35             sendRows();    // Send rows LB and UB to neighbors
36             receiveRows(); // Receive rows LB-1 and UB+1
37             // Calculate red fields in local rows LB ... UB
38
39             sendRows();    // Send rows LB and UB to neighbors
40             receiveRows(); // Receive rows LB-1 and UB+1
41             // Calculate black fields in local rows LB ... UB
42
43         } while (....)
44     }
45 }
```

Figure 4.2: Code skeleton for SOR, implementation for single cluster.

```java
1  public class Sender extends Thread {
2      private BinIntr dest;
3      private double[] row;
4
5      public Sender(BinIntr dest, double[] row) {
6          this.dest = dest;
7          this.row = row;
8      }
9
10     public void run() {
11         dest.put(row); // The RMI.
12     }
13 }
14
15 public class SOR {
16     private BinIntr leftBin, rightBin; // Remote bins of neighbors.
17     private Bin myLeftBin, myRightBin; // My own bin objects.
18
19     public void sendRows() {
20         if (leftBoundary) {  // Am I at a cluster boundary?
21             new Sender(leftBin, matrix[LB]).start(); // Async send.
22         } else {
23             leftBin.put(matrix[LB]); // synchronous RMI.
24         }
25
26         // Same for row UB to right neighbor ...
27     }
28
29     public void receiveRows() {
30         // same as in single-cluster implementation
31     }
32
33     public void run() {
34         do {    // do red/black SOR on the interval LB .. UB
35           sendRows();    // Send rows LB and UB to neighbors
36           Calculate red fields in local rows LB+1 ... UB-1
37           receiveRows(); // Receive rows LB-1 and UB+1
38           Calculate red fields in local rows LB and UB
39
40           sendRows();    // Send rows LB and UB to neighbors
41           Calculate black fields in local rows LB+1 ... UB-1
42           receiveRows(); // Receive rows LB-1 and UB+1
43           Calculate black fields in local rows LB and UB
44         } while (....)
45     }
46 }
```

Figure 4.3: Code skeleton for SOR, implementation for wide-area system.

clusters × CPUs	optimization	speedup	total	computation	sendRows	receiveRows
1 × 16	no	15.2	76888	75697	419	772
4 × 16	no	37.5	30973	18397	2088	10488
4 × 16	yes	53.9	21601	17009	4440	152
1 × 64	no	60.6	19091	17828	754	509

Table 4.2: Performance breakdown for SOR, average times per machine in milliseconds.

shown in Figure 4.3. During the RMI, the original SOR process can continue computing, so communication over the wide-area network is overlapped with computation. For communication within a cluster, the overhead of extra thread-switches slightly outweighs the benefits, so only inter-cluster communication is handled asynchronously (see the method *sendRows*). The optimization is thus purely applicable for wide-area systems, but does not improve performance on a single cluster.

The performance of the SOR program is shown in Figure 4.1. We ran a problem with a grid size of 4096 × 4096 and a fixed number of 64 iterations. The program obtains a high efficiency on a single cluster (a speedup of 60.6 on 64 processors). Without the optimization, SOR on the wide-area system achieves a speedup of only 37.5 on 4 × 16 processors. With the latency-hiding optimization, the speedup increases to 53.9, which is quite close to the speedup on a single 64-node cluster. Latency hiding thus is very effective for SOR. Table 4.2 presents a performance breakdown. It shows the total execution time in the *run* method, the time spent computing, and the time spent in the *sendRows* and *receiveRows* methods. The times in the table are the average values over all machines. Comparing the two runs with 4 × 16 machines shows the effectiveness of our optimization. With split-phase communication, the time spent in *receiveRows* is reduced dramatically. Communication is effectively overlapped with computation, reducing the time processors have to wait for boundary rows to arrive. The price for this gain is the creation of new threads for asynchronous sending. This is why the optimized version of *sendRows* takes about twice as much time as its unoptimized (non-threaded) counterpart. In total, the split-phase communication saves about 9.5 seconds compared to the unoptimized version.

4.2.2   All-pairs Shortest Paths Problem

The All-pairs Shortest Paths (ASP) program finds the shortest path between any pair of nodes in a graph, using a parallel version of Floyd's algorithm (see Section 2.7.2). The program uses a distance matrix that is divided row-wise among the available processors. At the beginning of iteration k, all processors need the value of the kth row of the matrix. The most efficient method for expressing this communication pattern would be to let the processor containing this row (called the owner) broadcast it to all the others. Unfortunately, Java RMI does not support broadcasting, so this cannot be expressed directly in Java. Instead, we simulate the broadcast with a spanning tree algorithm implemented using RMIs and threads.

The skeleton of a single-cluster implementation is shown in Figure 4.4. The interface *AspIntr* is omitted for brevity (it marks *transfer* as a remote method). All processes execute the *run* method of the class *Asp*. For broadcasting, they collectively call their

```
1  public class Sender extends Thread {
2      private AspIntr dest;
3      private int k, owner, row[];
4      private boolean filled = false;
5
6      synchronized void put(Asp dest, int[] row, int k, int owner) {
7          while(filled) wait();
8          this.dest = dest; this.row = row;
9          this.k = k; this.owner = owner;
10         filled = true;
11         notifyAll(); // Wake up thread waiting for row to arrive.
12     }
13
14     synchronized void send() {
15         while(!filled) wait();
16         dest.transfer(row, k, owner);  // do RMI to a child
17         filled = false;
18         notifyAll(); // Wake up thread waiting for row to arrive.
19     }
20
21     public void run() { while(true) send(); }
22 }
23
24 public class Asp extends UnicastRemoteObject Implements AspIntr {
25     private int[][] tab; // The distance table.
26     private AspIntr left, right; // My children in the broadcast tree.
27     private Sender leftSender, rightSender; // Threads for child RMIs.
28
29     public synchronized void transfer(int[] row, int k, int owner) {
30         if(left != null) leftSender.put(left, row, k, owner);
31         if(right != null) rightSender.put(right, row, k, owner);
32         tab[k] = row;
33         notifyAll();  // Wake up thread waiting for row to arrive.
34     }
35
36     public synchronized void broadcast(int k, int owner) {
37         if(myRank == owner) transfer(tab[k], k, owner);
38         else while (tab[k] == null) wait();  // Wait until row arrives.
39     }
40
41     public void run() {  // Computation part.
42         for (int k = 0; k < n; k++) {
43             broadcast(k, owner(k));
44             // Recompute my rows.
45         }
46     }
47 }
```

Figure 4.4: Code skeleton for ASP, implementation for single cluster.

```
1  class Sender extends Thread {} // Same as single—cluster version.
2
3  public class Asp extends UnicastRemoteObject Implements AspIntr {
4      private int[][] tab; // The distance table.
5      private AspIntr[] coordinators; // Remote cluster coordinators.
6      private Sender[] waSenders; // Threads for wide—area sends.
7
8      public synchronized void transfer(int[] row, int k, int owner) {
9          // Same as in single—cluster implementation.
10     }
11
12     public synchronized void broadcast(int k, int owner) {
13         if(I am the owner) {
14             // Separate thread for each cluster coordinator.
15             for(int i=0; i<nrClusters; i++) {
16                 waSenders[i].put(coordinators[i], row, k, owner);
17             }
18         } else {
19             while (tab[k] == null) wait();
20         }
21     }
22
23     public void run() {  // computation part
24         // Same as in single—cluster implementation.
25     }
26 }
```

Figure 4.5: Code skeleton for ASP, implementation for wide-area system.

broadcast method. Inside *broadcast*, all threads except the row owner wait until they re-
ceive the row. The owner initiates the broadcast by invoking *transfer*, which arranges all
processes in a binary tree topology. Such a tree broadcast is quite efficient inside clusters
with fast local networks. *transfer* sends the row to its left and right children in the tree,
using threads of class *Sender*. A *Sender* calls *transfer* on its destination node which recur-
sively continues the broadcast. For high efficiency, sending inside the binary tree has to be
performed asynchronously (via threads) because otherwise all intermediate nodes would
have to wait until the RMIs of the whole successive forwarding tree are completed. As
shown in Figure 4.1, ASP using the binary tree broadcast achieves almost linear speedup
when run on a single cluster. With a graph of 2500 nodes, it obtains a speedup of 57.9 on
a 64-node cluster.

A binary tree broadcast obtains poor performance on the wide-area system, causing
the original ASP program to run much slower on four clusters than on a single 16-node
cluster (see Figure 4.1). The reason is that the spanning tree algorithm does not take the
topology of the wide-area system into account, and therefore sends the same row multiple
times over the same wide-area link.

To overcome this problem, we implemented a wide-area optimized broadcast similar
to the one used in the MagPIe collective communication library [89]. With the optimized
program, the broadcast data is forwarded to all other clusters in parallel, over different

clusters × CPUs			optimization	speedup	total	computation	broadcast
1	×	16	no	15.6	230173	227908	2265
4	×	16	no	2.5	1441045	57964	1383081
4	×	16	yes	24.3	147943	56971	90972
1	×	64	no	57.9	61854	57878	3976

Table 4.3: Performance breakdown for ASP, average times per machine in milliseconds.

wide-area links. We implement this scheme by designating one of the *Asp* processes in each cluster as a *coordinator* for that cluster. The broadcast owner asynchronously sends the rows to each *coordinator* in parallel. This is achieved by one dedicated thread of class *Sender* per cluster. Using this approach, each row is only sent once to each cluster. Due to the asynchronous send, all wide-area connections can be utilized simultaneously. Inside each cluster, a binary tree topology is used, as in the single-cluster program. The code skeleton of this implementation is shown in Figure 4.5.

As shown in Figure 4.1, this optimization significantly improves ASP's application performance and makes the program run faster on four 16-node clusters than on a single 16-node cluster. Nevertheless, the speedup on four 16-node clusters lags far behind the speedup on a single cluster of 64 nodes. This is because each processor performs several broadcasts, for different iterations of the *k*-loop (see the *run* method). Subsequent broadcasts from different iterations wait for each other. A further optimization therefore would be to pipeline the broadcasts by dynamically creating new *Sender* threads (one per cluster per broadcast), instead of using dedicated threads. However, this would require a large number of dynamically created threads, even increasing with the problem size. This problem can be alleviated by using a pool of threads of a fixed size N, thus allowing N messages to be pipelined. If a truly asynchronous RMI would be available, the excessive use of additional threads could be completely avoided and the overhead related to thread creation and thread switching would also disappear.

Table 4.3 shows a performance breakdown for ASP. It shows the total execution time in the *run* method, the time spent computing, and the time spent in *broadcast*. The times in the table are the average values over all machines. Comparing the two runs with 4×16 machines, it can be seen that the wide-area optimization saves most of the communication costs of the unoptimized version. But even with the optimization, wide-area communication of 4×16 machines takes much more time, compared to 64 machines in a single cluster.

4.2.3   The Traveling Salesperson Problem

The Traveling Salesperson Problem (TSP) computes the shortest path for a salesperson to visit all cities in a given set exactly once, starting in one pre-specified city (see Section 2.7.7). We use a branch-and-bound algorithm, which prunes a large part of the search space by ignoring partial routes that are already longer than the current best solution. The program is parallelized by distributing the search space over the different processors. Because the algorithm performs pruning, however, the amount of computation needed for each sub-space is not known in advance and varies between different parts of the search

```
1  public interface MinIntr extends java.rmi.Remote {
2      public synchronized void update(int minimum);
3  }
4
5  public class Minimum extends UnicastRemoteObject implements MinIntr {
6      private int minimum; // The minimum value itself.
7      private Minimum[] PeerTable;   // Table of peers.
8
9      public synchronized void update(int minimum) {
10         if (minimum < this.minimum) {
11             this.minimum = minimum;
12         }
13     }
14
15     public synchronized void set(int minimum) {
16         if (minimum < this.minimum) {
17             this.minimum = minimum;
18
19             // notify all peers
20             for (int i=0; i < PeerTable.length; i++) {
21                 PeerTable[i].update(minimum); // RMI
22             }
23         }
24     }
25 }
```

Figure 4.6: Code skeleton for TSP, update of current best solution.

space. Therefore, load balancing becomes an issue. In single-cluster systems, load imbalance can easily be minimized using a centralized job queue. In a wide-area system, this would generate much wide-area communication. As wide-area optimization we implemented one job queue per cluster. The work is initially equally distributed over the queues. Whenever the number of jobs in a cluster queue drops below a low-water mark, wide-area steals are initiated. Job stealing between the cluster queues balances the load during run time without excessive wide-area communication. The job queues can be accessed over the network using RMI. Each job contains an initial path of a fixed number of cities; a processor that executes the job computes the lengths of all possible continuations, pruning paths that are longer than the current best solution. Each processor runs one *worker* thread that repeatedly fetches jobs from the job queue of its cluster, using an RMI called *get*, and executes the job, until all work is finished.

The TSP program keeps track of the best solution found so far, which is used to prune part of the search space. Each worker contains a copy of this value in an object of class *Minimum*. If a worker finds a better complete route, it does a sequence of RMIs to all peer workers to update their copies. Therfore, the *Minimum* values are declared as remote objects. The implementation shown in Figure 4.6 is straightforward and will not scale to a large numbers of workers. As minimum updates happen very infrequently, we found that a (cluster-aware) spanning tree broadcast was not necessary. Using a 17-city problem, we counted as few as 7 updates during the whole run of 290 seconds on 64 machines.

					time			*get* calls	
clusters	×	CPUs	optimized	speedup	total	computation	*get*	number	time per *get*
1	×	16	no	16.0	509768	509696	72	211	0.3
4	×	16	no	60.7	131763	127184	4579	53	86.4
4	×	16	yes	61.1	128173	127904	269	53	5.1
1	×	64	no	60.9	129965	127270	2695	53	50.9
1	×	64	yes	61.7	128144	128094	50	53	0.9

Table 4.4: Performance breakdown for TSP, average times per machine in milliseconds.

The performance for the TSP program on the wide-area DAS system is shown in Figure 4.1, using a 17-city problem. The run time of our TSP program is strongly influenced by the actual job distribution which results in different execution orders and hence in different amounts of routes that can be pruned. To avoid this problem, the results presented in Figure 4.1 and Table 4.4 have been obtained with the deterministic version of TSP (see Section 2.7.7); run time differences are thus only due to communication behavior.

As can be seen in Figure 4.1, the speedup of TSP on the wide-area system is only slightly inferior than on a single 64-node cluster. Our wide-area optimized version is even slightly faster than the unoptimized version on a single, large cluster. This is presumably because there is less contention on the queue objects when the workers are distributed over four queues. To verify this assumption, we also used our optimized version with four queues with 1×64 machines and obtained another slight performance improvement compared to a single queue and 1×64 machines. Table 4.4 supports this presumption. It presents the speedups, the average time spent by the worker threads in total, while computing, and while getting new jobs. It also shows the average number of jobs processed per worker and the average time per *get* operation. The speedups are computed for the whole parallel application (the maximum time over all processes), while the time values in the table are averaged over the worker threads.

On a single cluster with 64 machines, the average time per *get* operation is two orders of magnitude higher than on a cluster with only 16 machines. This shows that contention at the machine that owns the queue is in fact a problem. The total time spent in the *get* method is very low compared to the computation time which explains the high speedup values.

4.2.4   Iterative Deepening A*

Iterative Deepening A* is another combinatorial search algorithm, based on repeated depth-first searches (see Section 2.7.8). IDA* tries to find a solution to a given problem by doing a depth-first search up to a certain maximum depth. If the search fails, it is repeated with a higher search bound, until a solution is found. The search depth is initialized to a (tight) lower bound of the solution. IDA* is a branch-and-bound algorithm that uses pruning to avoid searching useless branches.

We wrote a parallel IDA* program in Java for solving the 15-puzzle (the sliding tile puzzle). IDA* is parallelized by searching different parts of the search tree concurrently. The program uses a more advanced load balancing mechanism than TSP, based on work

clusters × CPUs	optimization	speedup	total	time computation	get	jobs stolen local	remote
1 × 16	no	15.6	77384	75675	1709	69	
4 × 16	no	50.1	23925	19114	4811	46	15
4 × 16	yes	52.5	22782	19098	3684	62	8
1 × 64	no	55.4	21795	19107	2688	70	

Table 4.5: Performance breakdown for IDA*, average times per machine in milliseconds.

stealing. Each machine maintains its *own* job queue, but machines can get work from other machines when they run out of jobs. Each job represents a node in the search space. When a machine has obtained a job, it first checks whether it can prune the node. If not, it expands the node by computing the successor states (children) and stores these in its local job queue. To obtain a job, each machine first looks in its own job queue; if it is empty it tries the job queues of some other, randomly selected machines. We implemented one wide-area optimization: to avoid wide-area communication for work stealing, each machine first tries to steal jobs from ^{2}log(n) machines in its own cluster (where n is the number of machines in the local cluster). Only if that fails, the work queues of remote clusters are accessed. In each case, the same mechanism (RMI) is used to fetch work, so this heuristic is easy to express in Java. In Chapter 6, we will discuss a better way to optimize work stealing for wide-area systems (in the context of the Satin system).

Figure 4.1 shows the speedups for the IDA* program. The program takes about 5% longer on the wide-area DAS system than on a single cluster with 64 nodes. The communication overhead is due to work-stealing between clusters and to the distributed termination detection algorithm used by the program. The gain of the wide-area optimization is small in this case. For obtaining meaningful results across various parallel configurations, our IDA* implementation searches all solutions of equal depth for a given puzzle. Table 4.5 shows a performance breakdown for IDA*. The times presented are averages over all machines, showing the times spent in total, while computing, and while getting new jobs. Comparing the two runs with 4 × 16 machines it can be seen that our wide-area optimization is effective, but has only a minor impact on the total completion time. This is because job stealing occurs infrequently. The numbers of actually stolen jobs shown in the table are also average values per machine. It can be seen that the optimization helps reducing wide-area communication by reducing the number of jobs stolen from remote clusters.

4.3 Alternative Java-centric Grid Programming Models

We have discussed the implementation and wide-area optimization of four parallel applications using the RMI model. RMI supports transparent invocation of methods on remote objects and thus is a natural extension of Java's object model to distributed memory systems. We will now discuss alternative programming models and we compare them to RMI in terms of expressiveness (ease of programming) and implementation efficiency. We will discuss replicated objects [12, 103, 114, 164], JavaSpaces [60], and MPI for Java [34].

Some of these alternative models have been implemented in the Manta system. We refer to [110] and [122] for details and application measurements. Experiences in wide-area parallel programming using replicated objects and MPI in combination with other languages (Orca and C) are described elsewhere [13, 89, 132].

RMI

With RMI, parallel applications strictly follow Java's object-oriented model in which client objects invoke methods on server objects in a location-transparent way. Each remote object is physically located at one machine. Although the RMI model hides object remoteness from the programmer, the actual object location strongly impacts application performance.

Replication

From the client's point of view, object replication is conceptually equivalent to the RMI model. The difference is in the implementation: objects may be physically replicated on multiple processes. The advantage is that read-only operations can be performed locally, without any communication. The disadvantage is that write operations become more complex and have to keep object replicas consistent. Manta offers an object replication scheme called RepMI, which is described in more detail in [111].

JavaSpaces

JavaSpaces adapts the Linda model [65] to the Java language. Communication occurs via shared data *spaces* into which *entries* (typed collections of Java objects) may be written. Inside a space, entries may not be modified, but they may be read or removed (taken) from a space. A reader of an entry provides a *template* that matches the desired entry type and also desired object values stored in the entry. Wildcards may be used for object values. Additionally, a space may notify an object whenever an entry matching a certain template has been written. *Space* objects may be seen as objects that are remote to all communicating processes; read and write operations are implemented as RMIs to space objects. JavaSpaces also supports a transaction model, allowing multiple operations on space objects to be combined in a transaction that either succeeds or fails as a whole. This feature is especially useful for fault-tolerant programs.

MPI

With the Message Passing Interface (MPI) language binding to Java, communication is expressed using message passing rather than remote method invocations. Processes send messages (arrays of objects) to each other. Additionally, MPI defines collective operations in which all members of a process group collectively participate; examples are broadcast and related data redistributions, reduction computations (e.g., computing global sums), and barrier synchronization. A drawback of MPI is that it does not integrate nicely with Java's object-oriented model [110].

	RMI	replication	JavaSpaces	MPI
send	synchronous	synchronous	synchronous	sync and async
receive	implicit	implicit	explicit and implicit	explicit
collective communication	no	broadcast	no	yes

Table 4.6: Aspects of programming models

4.3.1 Comparison of the Models

To compare the suitability of the models for wide-area parallel computing, we study three important aspects. Table 4.6 summarizes this comparison. In general, we assume that the underlying programming system (like Manta) exposes the physical distribution of the hierarchical wide-area system. This information may be used either by application programs (as with Manta) or by programming platforms designed for wide-area systems (e.g., the MagPIe library [89]).

Synchronous vs. asynchronous communication

Remote method invocation is a typical example of synchronous communication. Here, the client has to wait until the server object has returned the result of the invoked method. This enforces synchronization of the client with another, possibly remote process. With *asynchronous* communication, the client may immediately continue its operation after the communication has been initiated. It may later check or wait for completion of the communication operation. Asynchronous communication is especially important for wide-area computing, where it can be used to hide the high message latencies by overlapping communication and computation. It is important to note that RMIs can never be executed asynchronously, even when the return type is *void*, because RMIs can throw an exception.

MPI provides asynchronous sending and receiving. The other three models, however, rely on synchronous method invocation so applications have to simulate asynchronous communication using multithreading. For local-area communication, the corresponding overhead for thread creation and context switching may exceed the cost of synchronous communication. To avoid the high overhead of thread creation for local messages, the optimized code for SOR gets rather complicated, as has been shown in Section 4.2.1. The broadcast implementation in ASP also requires asynchronous communication, both in the original (single cluster) and wide-area optimized version. With synchronous RMI, a broadcast sender would have to wait for the whole spanning tree to complete. The TSP efficiency could also be improved by treating local and remote communication differently when updating the value of the current best solution. Fortunately, TSP is less sensitive to this problem because these updates occur infrequently. In IDA*, work stealing is always synchronous but fortunately infrequent. So, IDA* is hardly affected by synchronous RMI.

In conclusion, for wide-area parallel computing on hierarchical systems, directly supporting asynchronous communication (as in MPI) is easier to use and more efficient than using synchronous communication and multithreading.

Explicit vs. implicit receipt

A second issue is the way in which messages are received and method invocations are served. With RMI and with replicated objects, method invocations cause upcalls on the server side, which is a form of implicit message receipt. MPI provides only explicit message receipt, making the implementations of TSP and IDA* much harder. Here, incoming messages (for updating the global bound or for job requests) have to be polled for by the applications. This complicates the programs and makes them also less efficient, because finding the right polling frequency (and the best places in the applications to put polling statements at) is difficult [12, 98]. With hierarchical wide-area systems, polling has to simultaneously satisfy the needs of LAN and WAN communication, making the problem of finding the right polling frequency even harder. With programming models like RMI that support implicit receipt, application-level polling is not necessary.

JavaSpaces provides explicit operations for reading and removing elements from a global space. Additionally, a space object may notify a potential reader whenever an entry has been written that matches the reader's interests. Unfortunately, this notification simply causes an upcall at the receiver side which in turn has to actually perform the read or take operation. This causes a synchronous RMI back to the space object. The additional overhead can easily outweigh the benefit of implicit receipt, especially when wide-area networks are used.

Point-to-point vs. collective communication

Wide-area parallel programming systems can ease the task of the programmer by offering higher-level primitives that are mapped easily onto the hierarchical structure of the wide-area system. In this respect, we found MPI's collective operations (e.g., broadcast and reduction) to be of great value. The MagPIe library [89] optimizes MPI's collective operations for hierarchical wide-area systems by exploiting knowledge about how groups of processes interact. For example, a broadcast operation defines data transfers to all processes of a group. MagPIe uses this information to implement a broadcast that optimally utilizes wide-area links; e.g., it takes care that data is sent only once to each cluster. Replicated objects that are implemented with a write-update protocol [12] can use broadcasting for write operations, and thus can also benefit from wide-area optimized broadcast.

Programming models without group communication (like RMI and JavaSpaces) cannot provide such wide-area optimizations inside a runtime system. Here, the optimizations are left to the application itself, making it more complex. The broadcast implemented for ASP is an example of such an application-level optimization that could be avoided by having group communication in the programming model. The MagPIe implementation of ASP [89], for example, is much simpler than the (wide-area optimized) Java version. Similarly, TSP's global bound could be updated using a pre-optimized group operation.

4.3.2   Summary of Alternative Programming Models

The remote method invocation model provides a good starting point for wide-area parallel programming, because it integrates communication into the object model. Another

benefit is RMI's implicit receipt capability by which RMIs are served using upcalls. This contributes to expressiveness of the programming model as well as to program efficiency, especially with wide-area systems.

To be fully suited for parallel systems, RMI needs two extensions. The first one is asynchronous method invocation, which is especially important for wide-area systems. The second extension is the implementation of collective communication operations that transfer the benefits of MPI's collective operations into Java's object model.

Although promising at first glance, JavaSpaces does not really make wide-area parallel programming easier, because operations on space objects are neither asynchronous nor collective. As fault-tolerance becomes more important in wide-area computing, the JavaSpaces model along with its transaction feature may receive more attention.

4.4   Related Work

Most work on grid computing focuses on how to build the necessary infrastructure [5, 56, 70, 139]. In addition, research on parallel algorithms and applications is required, since the bandwidth and latency differences in a grid computer can easily exceed three orders of magnitude [52, 56, 70, 132]. Coping with such a large non-uniformity in the interconnect complicates application development. The ECO system addresses this problem by automatically generating optimized communication patterns for collective operations on heterogeneous networks [108]. The AppLeS project favors the integration of workload scheduling into the application level [16].

Earlier research experiments with optimizing parallel programs for hierarchical interconnects, by changing the communication structure [13], or by trading communication for computations, as is done with wide-area Transposition-Driven Scheduling (TDS) [144]. The fact that synchronous RPC is not sufficient to efficiently express all communication patterns was also recognized by Amoeba [160]. The sensitivity of optimized programs to large differences in latency and bandwidth between the LAN and WAN has been studied by Plaat et al. [132]. The outcome was that very different wide-area optimizations are needed for different applications. Based on this experience, collective communication operations, as defined by the MPI standard, were implemented by Kielmann et al. [89], resulting in improved application performance on wide area systems. Some of the ideas of this earlier work have been applied in our wide-area Java programs.

4.5   Conclusion

We have described our experiences in using a high-performance Java RMI system that runs on a geographically distributed (wide-area) system. The goal of our work was to obtain actual experience with a Java-centric approach to grid computing and to investigate the usefulness of RMI for distributed supercomputing. The Java system we have built is highly transparent: it provides a single communication primitive (RMI) to the user, even though the implementation uses several communication networks and protocols.

We have implemented several parallel applications on this system, using Java RMI for communication. In general, the RMI model was easy to use. To obtain good performance,

the programs take the hierarchical structure of the wide-area system into account and minimize the amount of communication (RMIs) over the slow wide-area links. With such optimizations in place, the programs can effectively use multiple clusters, even though they are connected by slow links.

A problem is, however, that each application had to be optimized individually to reduce the utilization of the scarce wide-area bandwidth or to hide the large wide-area latency. We found that different applications may require very different wide-area optimizations. In general, it is hard for a programmer to manually optimize parallel applications for hierarchical systems.

We compared RMI with other programming models, namely object replication, JavaSpaces, and MPI for Java. We identified several shortcomings of the RMI model. In particular, the lack of asynchronous communication and of a broadcast mechanism complicates programming. MPI offers both features (and further useful collective operations) but lacks RMI's clean object-oriented model. Object replication is closer to pure RMI but offers only broadcast-like object updating. Object replication and collective operations are implemented in the Manta system, and are described in more detail in [110, 111, 122].

We conclude that Java RMI is a useful building block for grid applications. It is relatively straightforward to write distributed supercomputing applications with RMI, and it is also possible to implement (application-specific) wide-area optimizations. However, it is less straightforward to write *efficient* distributed supercomputing applications with RMI, because of several limitations of the RMI model. The most important drawbacks of RMI are the lack of asynchronous communication, which is especially important with the high wide-area latencies, and multicast support. This functionality can be implemented on top of RMI, using threads. For instance, the SOR application implements asynchronous communication by letting a separate thread execute the RMI, and ASP implement multicast by building a spanning tree with RMIs and threads. However, the use of threads has considerable overhead, which has to be taken into account. In SOR, for example, the use of threads for local communication only slows things down. With an asynchronous communication primitive, the use of threads could be avoided altogether.

In the remainder of this thesis, we take an alternative approach. We describe a system called Satin, which tries to run *one specific class* of applications (divide-and conquer algorithms) efficiently on wide-area systems without special wide-area optimizations at the application level. Satin implements some of the lessons learned with RMI (e.g., latency hiding, avoiding threading overhead). However, Satin implements these features inside the runtime system, making them transparent for the programmer.

Chapter 5

Satin: Divide-and-Conquer-Style Parallel Programming

```
Divide et impera. -Divide and rule. (Divide and conquer.)

- The motto of Niccolo Machiavelli, but was also used by
Philip of Macedon, Louis XI of France and of Austria,
Polybius, Bossuet, and Montesquieu.
```

The ultimate goal of our work is to create a programming environment in which parallel applications for hierarchical systems are easy to implement. In the previous chapter, we concluded that RMI is useful in this area, but still has several shortcomings. Ideally, the application programmer should not have to implement different wide-area optimizations for each application by hand. One possible solution of this problem is investigated in detail in this chapter and the next one. We chose one specific class of problems that are inherently hierarchical, divide-and-conquer algorithms, and implemented an efficient compiler and runtime system that apply wide-area optimizations automatically.

Divide-and-conquer algorithms work by recursively dividing a problem into smaller subproblems. This recursive subdivision goes on until the remaining subproblem becomes trivial to solve. After solving subproblems, their results are recursively recombined until the final solution is assembled. This process is described in more detail in Section 2.5.

Divide-and-conquer applications may be parallelized by letting different processors solve different subproblems. These subproblems are often called *jobs* in this context. Generated jobs are transferred between processors To balance the load in the computation.

The divide-and-conquer model lends itself well for hierarchically-structured systems because tasks are created by recursive subdivision. This leads to a task graph that is hierarchically structured, and which can be executed with excellent communication locality, especially on hierarchical platforms. Of course, there are many kinds of applications that do not lend themselves well to a divide-and-conquer algorithm. However, we (and others) believe the class of divide-and-conquer algorithms to be sufficiently large to justify its deployment for hierarchical wide-area systems. Computations that use the divide-and-

conquer model include geometry procedures, sorting methods, search algorithms, data classification codes, n-body simulations and data-parallel numerical programs [174].

There is currently much interest in divide-and-conquer systems for parallel programming [14, 22, 64, 100, 146]. An example of such a system is Cilk [22], which extends C with divide-and-conquer primitives. Cilk runs these annotated C programs efficiently in parallel, but is mainly targeted at shared-memory machines. CilkNOW [24] is a distributed memory version of Cilk. Atlas [14], a set of Java classes, is a divide-and-conquer system designed for distributed memory machines. Its primitives have a high overhead, however, so it runs fine-grained parallel programs inefficiently.

We introduce a new system, called Satin, designed specifically for running divide-and-conquer programs on distributed memory systems (and ultimately on wide-area grid computing systems). Satin (as the name suggests) was inspired by Cilk. In Satin, single-threaded Java programs are parallelized by annotating methods that can run in parallel. Our ultimate goal is to use Satin for distributed supercomputing applications on hierarchical wide-area systems (e.g., the DAS [11]), and more general, on the grid. We think that the divide-and-conquer model will map efficiently on such systems, as the model is also hierarchical. In this chapter, however, we focus on the implementation of Satin on a single, local cluster computer. The next chapter deals with Satin on wide-area systems. In contrast to Atlas, Satin is designed as a compiler-based system in order to achieve high performance. Satin is based on the Manta native compiler, which supports highly efficient serialization and communication (see Chapter 3). Parallelism is achieved in Satin by running different spawned method invocations on different machines. The system load is balanced by the Satin runtime system.

One of the key differences between Satin and earlier divide-and-conquer systems for distributed memory systems (such as CilkNOW), is that Satin makes a deep copy of the parameters to spawned methods, while the earlier systems make a shallow copy. In this sense, the Satin parameter model is close to RMI. Making a deep copy of the parameters provides a more convenient programming model, as not only the parameters of a spawned method themselves are transferred to remote machines, but also *all data that can be reached (directly or indirectly) via those parameters*. This makes it possible to pass arbitrary graphs of data structures as a parameter to a spawned method.

Satin extends Java with two simple Cilk-like primitives for divide-and-conquer programming. The Satin compiler and runtime system cooperate to implement these primitives efficiently on a distributed system, using work stealing to distribute the jobs. Satin reduces the overhead of local jobs using on-demand serialization, which avoids copying and serialization of parameters for jobs that are not stolen. Without this optimization, the overhead of making a (deep) copy of all parameters in the local case would have been prohibitive, even with Manta's efficient, compiler-generated serialization.

The contributions we make in this chapter are the following:

- we show that, using invocation records, it is possible to enable the on-demand serialization of parameters to spawned method invocations;

- we demonstrate that a careful choice of parameter semantics makes this optimization possible;

- we integrate divide-and-conquer cleanly into Java, without altering the language itself.

- we solve some problems that are introduced by this integration (e.g., by garbage collection).

- we describe how the Satin implementation can run parallel divide-and-conquer programs efficiently on a cluster of workstations;

- we demonstrate that, even though Satin does not provide shared memory, replicated objects can be used to efficiently implement shared data.

In the remainder of this chapter, we will first define the semantics of the Satin extensions (Section 5.1). Next, in Section 5.2, we describe the implementation of the Satin runtime system, as well as the extensions we made to the Manta compiler. In Section 5.3, we analyze the performance of some application kernels, Finally, we describe related work (Section 5.4) and draw our conclusions in Section 5.5.

5.1 The Programming Model

Satin's programming model is an extension of the single-threaded Java model. Satin programmers thus need not use Java's multithreading and synchronization constructs or Java's Remote Method Invocation mechanism, but can use the much simpler divide-and-conquer primitives described below. Early versions of Satin used several new keywords to implement the divide and conquer primitives [124]. The version described in this chapter uses special "marker" interfaces instead, so that the Java language is unaltered.

While the use of Java threads is not necessary with Satin, it is possible to combine Satin programs with Java threads. This can be useful for user interfaces, for instance. Furthermore, it is possible to use RMI [157], RepMI [111], GMI [110] or any other communication mechanism in combination with Satin.

5.1.1 Spawn and Sync

Parallel divide-and-conquer systems have at least two primitives: one to spawn work, and one to wait until the spawned work is finished. Cilk introduces new keywords into the C language to implement these primitives. Satin integrates cleanly into Java, and uses no language extensions. Instead, Satin exploits Java's standard mechanisms, such as inheritance and the use of marker interfaces (e.g., java.io.Serializable) to extend Java with divide-and-conquer primitives.

In Satin, a *spawn* operation is a special form of a method invocation. Methods that can be spawned are defined in Satin by tagging methods with a special marker interface. We will call such methods Satin methods. A call to a method that was tagged as spawnable is called a *spawned method invocation*. With a spawn operation, conceptually a new thread is started which will run the method (the implementation of Satin, however, eliminates thread creation altogether). The spawned method will run concurrently with the method that executed the spawn. The *sync* operation waits until all spawned calls in this method

```
1  interface FibInter extends satin.Spawnable {
2      public long fib(long n);
3  }
4
5  class Fib extends satin.SatinObject implements FibInter {
6      public long fib(long n) {
7          if(n < 2) return n;
8
9          long x = fib(n−1); // Spawns, because fib is
10         long y = fib(n−2); // tagged in FibInter.
11         sync();
12
13         return x + y;
14     }
15
16     public static void main(String[] args) {
17         Fib f = new Fib();
18         long res = f.fib(10);
19         f.sync();
20         System.out.println("Fib 10 = " + res);
21     }
22 }
```

Figure 5.1: *Fib*: an example divide-and-conquer program in Satin.

invocation are finished. The return values of spawned method invocations are undefined until a sync is reached. The assignment of the return values is not guaranteed to happen in the sync operation, but instead is done between the spawn and sync. All operations that exit a method (e.g., *return* or *throw*) are treated as an implicit sync operation.

The programmer can create an interface which extends the marker interface called *satin.Spawnable*, and define the signatures of methods that must be spawned. A class that spawns work must extend the special class *satin.SatinObject* to inherit the *sync* method. This mechanism closely resembles Java RMI (see 2.3.3). Spawned methods can throw exceptions, just like ordinary Java methods. We will describe the exception semantics in Chapter 8.

To illustrate the use of the spawn and sync operations, an example program is shown in Figure 5.1. This code fragment calculates Fibonacci numbers, and is a typical example of a divide-and-conquer program. The same program, but using Java threads instead of Satin's primitives, was discussed in Section 2.3.1. The Satin version of the program is much shorter and does not need the complex synchronization with Java's *wait* and *notify* operations. Note that the *Fib* program is a benchmark, and *not* a suitable algorithm for efficiently calculating the Fibonacci numbers. The work is split up into two pieces, *fib(n-1)* and *fib(n-2)*, which will then be recursively solved. Splitting the problem into smaller subproblems will continue until it can no longer be split up (i.e., n < 2). At that point, the answer for the subproblem is returned. Next, two solved subproblems are combined, in this case, by adding the results.

The program is parallelized just by tagging the *fib* method as a Satin method, using the special marker interface. The two subproblems will now be solved concurrently. Before

the results are combined, the method must wait until both subproblems have actually been solved, and have returned their value. This is done by the sync operation (which is inherited from *SatinObject*). A well known optimization in parallel divide-and-conquer programs is to make use of a threshold on the number of spawns. When this threshold is reached, work is executed sequentially. This approach can easily be programmed using Satin, and will be investigated in more detail in Section 5.3.1.

Satin does not provide shared memory, because this is hard to implement efficiently on distributed memory machines (like a DSM system). Moreover, our ultimate goal is to run Satin on wide-area systems, which clearly do not have shared memory. DSMs for wide-area systems are not (yet) feasible (see Section 2.3.4). Satin programmers can use Manta's replicated object system (RepMI) to implement shared data (see Section 5.2.5).

Because spawned method invocations may run on any (remote) machine, global variables that are defined on the machine that spawned the work may not be available at the site where the method is executed. The code of the spawned method should only access data that is stored on the machine that runs the work. Essentially, this means that Satin methods must not change global state. In pure Satin, the only way of communicating between jobs is via the parameters and the return value. In short, Satin methods must not have side-effects. The parameter passing mechanism, as described below, assures that all data that can be (directly or indirectly) accessed via parameters will be sent to the machine that executes the spawned method invocation. RMI, RepMI, or any other communication mechanism can be used in combination with Satin if this purely functional behavior is to restrictive. An example of how this can be done is shown in Section 5.2.5.

5.1.2 The Parameter Passing Mechanism

Since Satin does not provide shared memory, objects passed as parameters in a spawned call to a remote machine will not be available on that machine. Therefore, Satin uses *call-by-value* semantics when the runtime system decides that the method will be spawned remotely. This is semantically similar to the standard Java Remote Method Invocation (RMI) mechanism [167]. Call-by-value is implemented using Java's *serialization* mechanism, which provides a *deep copy* of the serialized objects [159]. For instance, when the first node of a linked list is passed as an argument to a spawned method invocation (or a RMI), the entire list is copied. Copying of a field can be avoided by using the standard Java *transient* modifier.

It is important to minimize the overhead for work that does not get transfered, but is executed by the machine that spawned the work, as this is the common case. For example, in almost all applications we have studied so far, at most 1 out of 150 jobs is transfered to a remote machine. Because copying all parameter objects (i.e., using call-by-value) in the local case would be prohibitively expensive, parameters are passed by reference when the method invocation is local. Therefore, the programmer cannot assume either call-by-value or call-by-reference semantics for Satin methods (normal methods are unaffected and have the standard Java semantics). It is therefore erroneous to write Satin methods that depend on the parameter passing mechanism. (A similar approach is taken in Ada for parameters of a structured type.)

5.1.3   Semantics of Sequential Satin Programs

An important characteristic of Satin is that its programs can be compiled with *any* Java compiler (including Manta), because Satin does not use language extensions. However, because standard Java compilers do not recognize Satin's special marker interface, the result is a sequential program. Only when the special Satin compiler (which is an extension of the Manta compiler) is used, parallel code is generated. The sequential version (i.e., the Satin program compiled with a normal Java compiler) produces the same result as the parallel Satin program. This is always true, because Satin does not specify the parameter passing mechanism. Using call-by-reference in all cases (as normal Java does) is thus correct.

5.2   The Implementation

The large majority of jobs will not be stolen, but will just run on the machine that spawned the work. Therefore, it is important to reduce the overhead that the Satin runtime system generates for such jobs as much as possible. The key problem here is that the decision whether to copy the parameters must be made at the moment the work is either executed or stolen, not when the work is generated. To be able to defer this important decision, Satin's runtime system uses invocation records, which will be described below. The large overhead for creating threads or building task descriptors (copying parameters) was also recognized in the lazy task creation work by Mohr et al. [116] and by Cilk [22].

5.2.1   Invocation Records

When a program executes a spawn operation, Satin redirects the method call to a stub. This stub creates an *invocation record* (see Figure 5.2), describing the method to be invoked, the parameters that are passed to the method, and a reference to where the method's return value (or exception) has to be stored. For primitive types, the value of the parameter is copied. For reference types (objects, arrays, interfaces), only a reference is stored in the record. In the example of Figure 5.2, a Satin method, called *foo*, is invoked with an integer, an array, and an object as parameters. The integer is stored directly in the invocation record, but for the array and the object, references are stored, to avoid copying these data structures. If the same parameter is passed to subsequent spawns, the invocation records will contain a pointer to the same physical object. This is shown with the *o* parameter in Figure 5.2.

  The compiler allocates space for a counter on the stack of all methods executing spawn operations. This counter is called the *spawn counter*, and counts the number of pending spawns, which have to be finished before this method can return. The *address* of the spawn counter is also stored in the invocation record. This way, the Satin runtime system can decrease the counter by one when a spawned method is finished.

  The stub that builds an invocation record for a spawned method invocation is generated by the Manta compiler, and is therefore very efficient, as no runtime type inspection is required. From an invocation record, the original call can be executed by pushing the

```
int foo(int i, double[] d, Object o);

int result = spawn foo(i, d1, o);
int result = spawn foo(i, d2, o);
int result = spawn foo(i, d3, o);
```
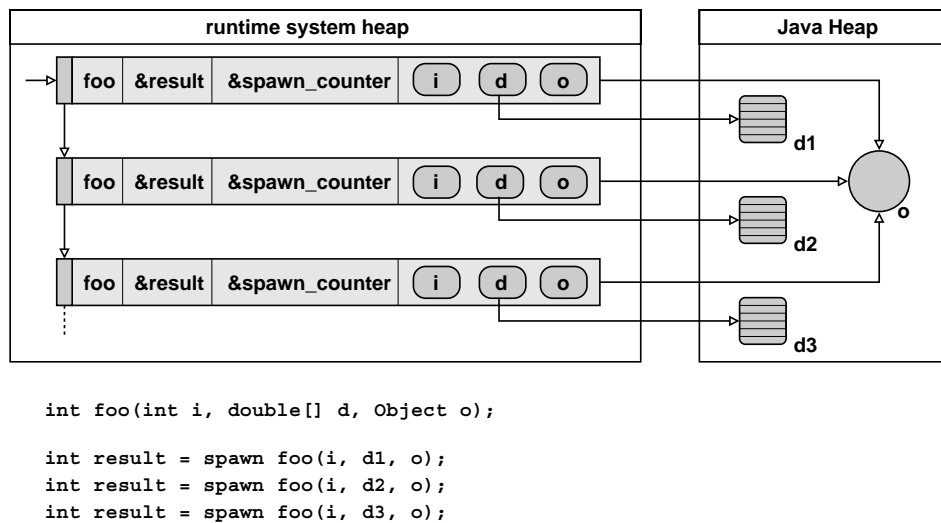
Figure 5.2: Invocation records in the job queue.

value of the parameters (which were stored in the record) onto the stack, and by calling the Java method.

The invocation record for a spawn operation is stored in a queue. The spawn counter (located on the stack of the invoking method) is incremented by one, indicating that the invoking method now has a pending spawned method invocation. The invoking method may then continue running. After the spawned method invocation has eventually been executed, its return value will be stored at the return address specified in the invocation record. Next, the spawn counter (the address of which is also stored in the invocation record) will be decremented by one, indicating that there now is one less pending spawn. The sync operation waits for the spawn counter to become zero. In the meantime, it executes work stored in the job queue. When this happens, there are no more pending spawned method invocations, so the sync operation is finished and the method may continue.

5.2.2 Serialization-on-Demand

Earlier divide-and-conquer systems for distributed memory systems (such as CilkNOW) make a shallow copy of the parameters of spawned calls. As there is no other means of sharing data, this means that the programmer must pack all data that is needed for the job into the parameters of the spawned call. When complicated data structures are used, these must be flattened by hand by the programmer. This can be done by packing the data into arrays or objects (structs) without pointers to other data.

Satin, however, makes a deep copy of the parameters to spawned methods. This is one of the key differences between Satin and the earlier systems. Making a deep copy of the

parameters provides a more convenient programming model, as not only the parameters of a spawned method themselves are transferred to remote machines, but also *all data that can be reached (directly or indirectly) via those parameters*. This makes it possible to pass arbitrary graphs of data structures as a parameter to a spawned method. Flattening complex data structures by hand is no longer needed. Another advantage is that this model is familiar to Java programmers, as Java RMI also makes deep copies of parameters.

Serialization is Java's mechanism to convert objects into a stream of bytes. This mechanism always makes a deep copy of the serialized objects: all references in the serialized object are traversed, and the objects they point to are also serialized. Java's serialization mechanism is described in more detail in Section 2.3.2. The serialization mechanism is used in Satin for marshalling the parameters to a spawned method invocation. Satin implements serialization on demand: the parameters are serialized only when the work is actually stolen. In the local case, no serialization is used, which is of critical importance for the overall performance. In the Manta system, the compiler generates highly-efficient serialization code. For each class in the system a so-called serializer is generated, which writes the data fields of an object of this class to a stream. When an object has reference fields, the serializers for the referenced objects will also be called. Furthermore, Manta uses an optimized protocol to represent the serialized objects in the byte stream. Manta's implementation of the serialization mechanism is described in more detail in Section 3.3.2.

5.2.3   The Double-Ended Job Queue and Work Stealing

The invocation records describing the spawned method invocations are stored in a double-ended job queue. Newly generated work is always inserted at the head of the queue. After the work has been put into the queue, the execution of the program continues behind the spawn operation. Eventually, a sync operation will be reached. At this point, the Satin runtime system starts executing work out of the job queue. Locally, work is always taken from the head of the queue, thus the queue is effectively used as a stack.

When a node runs out of work, it will start stealing work from other nodes. Idle nodes will poll remote queues for jobs, at the tail of the queue. The reason why local nodes execute work from the head of the queue, while remote nodes steal at the tail of the queue is that in this way large-grain jobs are stolen, reducing communication overhead [62].

This idea is explained in more detail by Figure 5.3. This figure shows the growing of the job queue in time, for the Fibonacci program shown earlier in Figure 5.1. The rectangles (the nodes in the job queue) are the invocation records. For every fib invocation, two new subproblems are spawned and inserted at the head of the queue. Then the sync operation is reached, and the job at the head of the queue is executed. This job will consecutively generate two new subproblems, and so on. It is inherent to the divide-and-conquer model that, in general, larger jobs are located towards the tail of the queue, as can be seen in the figure. This happens because all jobs are created by splitting larger jobs. However, jobs that are spawned within the same method are not necessarily sorted by size. Still, the trend is that the smallest jobs are at the head of the queue, while the largest jobs are at the tail. Therefore, remote nodes steal their work at the tail of the queue (not shown in figure 5.3). This way, they tend to steal the largest available job. This is
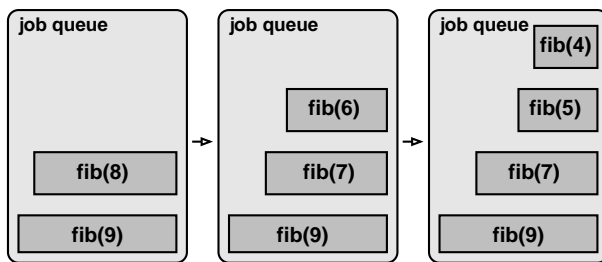
Figure 5.3: The job queue for the fib benchmark.

the desired behavior, because when a large job is stolen, the stealing node will not have to steal again soon, thus minimizing communication.

When the stolen job is finished, the return value will be serialized and sent back to the originating node. There, the return value is assigned, using the pointer to the result location that is kept in the invocation record. Further, the spawn counter is decreased, indicating that there is one less pending spawn.

The fact that there is only one thread stealing work from the head of the queue (the thread on the local machine), and multiple threads are stealing from the tail (the remote machines), makes it possible to use Dijkstra's protocol [46] for locking the queue. Using this protocol, locking can be avoided in the local case most of the time. When stealing work from the tail of the queue, locking is always needed.

For homogeneous (single cluster) systems, *Random Stealing* (RS) is known to achieve optimal load balancing. It is proven to be optimal in space, time and communication [23]. Therefore, Satin uses RS to balance the load of the computation on a single cluster of workstations. On wide-area systems, different load-balancing algorithms are needed. These are the subject of the next chapter. All measurements in this chapter are done on a single cluster, and use RS.

Satin's work stealing is implemented on top of the Panda communication library [12], using Panda's message passing and locking primitives. On the Myrinet network (which we use for our measurements), Panda is implemented on top of the LFC [19] network interface protocol. Satin uses the efficient, user-level locks that Panda provides for protecting the work queue.

5.2.4 Garbage Collection

An important feature of Java is that it supports garbage collection: objects that are no longer referred to are freed automatically. This has some impact on the Satin implementation. Satin stores the parameters to a spawned method invocation in an invocation record, and the method that executed the spawn will then continue. At this point, the parameters are no longer on the Java stack, but stored in the Satin runtime system, which is written in C. Therefore, the garbage collector, scanning the memory used by the Java program, could assume that object parameters passed to a spawned method invocation are

no longer in use. These parameters are then freed, while they are still needed to execute the spawned method. Satin solves this problem by registering the invocation records at the garbage collector, keeping parameter objects alive while they are referenced only via the invocation record. When a spawned method invocation returns, Satin unregisters the invocation record with the garbage collector.

5.2.5   Replicated Objects

Manta provides a replicated object system, called RepMI [111]. Satin can be used in combination with Manta's replicated objects. This way, the Satin language is more expressive, and Satin methods can change global state. RepMI uses function shipping to keep the replicas consistent. The replicated objects are method based, and it is not possible to directly modify fields in the replicated objects. Call-by-value is used to transfer parameters to the replicas. Satin is also method based, and does not specify whether call-by-value or call-by-reference is used. RepMI's objects thus integrate seamlessly into Satin's programming model. A more detailed description of RepMI can be found in [111].

Figure 5.4 shows an example of a Satin version of the traveling salesperson (TSP) program that uses a replicated object. Instead of a replicated object, using a remote object (via Java RMI) is also possible. The advantage of object replication compared to RMI is that methods which only read objects can be performed locally, without any communication. Only write operations cause communication across the set of replicas.

TSP can effectively prune the search space using a global minimum value, which represents the shortest solution found so far. If a partial solution is already longer than the global minimum, the rest of the path does not have to be expanded, and the search tree can be pruned. The measurements presented in the performance section do not use the replicated object. Instead, the minimum is set to the length of the solution. This way, all work that can be pruned is in fact ignored, and the program is completely deterministic. When a global minimum is used to prune the search space, the program becomes nondeterministic, making speedup measurements difficult.

Normally, when TSP is not used for benchmarking purposes, the minimum route is not known in advance. Then, the version that uses the replicated object is much more efficient. Even though there is some overhead introduced by reading and updating the replicated object, the pruning of work is more important. We will use the TSP example to investigate the performance of the replicated objects in Section 5.3.3.

5.3   Performance Evaluation

We evaluated Satin's performance using twelve application kernels. [1] The applications are described in more detail in Section 2.7. All measurements were performed on the cluster of the Distributed ASCI Supercomputer (DAS) that is located at the Vrije Universiteit. The hardware platform is described in Section 1.9.

---

[1]The Satin version used for the measurements in this chapter implemented Satin's operations with keywords instead of the special marker interface which the current version uses. However, this is only a syntactic difference, and has no impact on the performance numbers.

```
1  final class Minimum implements manta.runtime.Replicator {
2      int val = Integer.MAX_VALUE;
3
4      synchronized void set(int new_val) {
5          if(new_val < val) val = new_val;
6      }
7
8      synchronized int get() {
9          return val;
10     }
11 }
12
13 /* Search a TSP subtree that starts with initial route "path"
14    If partial route is longer than current best full route
15    then forget about it. */
16 void tsp(int hops, byte[] path, int length,
17          Minimum min, DistanceTable distance) {
18
19     int NTowns = distance.getSize();
20
21     // stop searching, this path is to long...
22     if (length >= min.get()) return;
23
24     // Found a route better than current best route, update minimum.
25     if (hops == NTowns) min.set(length);
26
27     // "path" is a partial route, call tsp recursively for subtrees.
28     int me = path[hops − 1]; // Last city of path so far.
29
30     // Try all cities that are not on the initial path.
31     for (int i = 0; i < NTowns; i++) {
32         int city = distance.getToCity(me, i);
33         if (city != me && !present(city, hops, path)) {
34             byte[] newpath = (byte[]) path.clone();
35             newpath[hops] = (byte) city;
36             int newLen = length + distance.getDist(me, i);
37
38             tsp(hops+1, newpath, newLen, min, distance); // Spawn.
39         }
40     }
41     sync();
42 }
```

Figure 5.4: A Satin example that uses a replicated object: TSP.

5.3.1 Spawn Overhead

An important indication of the performance of a divide-and-conquer system is the over-
head of the parallel application on one machine, compared to the sequential version of the
same application. The sequential version is obtained by compiling the Satin program with
the standard Manta compiler (i.e., with the Satin extensions turned off). The difference

parameters	time (μs)
void	1.64
1 int	1.80
2 int	1.94
3 int	2.13
4 int	2.30
5 int	2.43
1 float	1.81
1 double	1.80
1 long	1.81
1 Object	1.81

Figure 5.5: The cost of spawn opera-
tions.
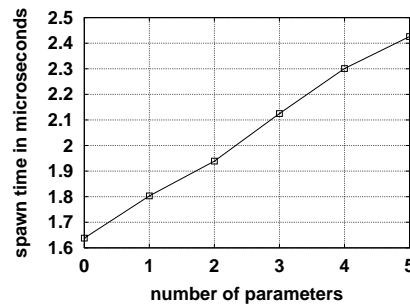
Figure 5.6: Cost of adding parameters to
spawn operations.

in run times between the sequential and parallel programs is caused by the creation, the
enqueuing and dequeuing of the invocation record, and the construction of the stack frame
to call the Java method.

This overhead factor is an important quantity, because it indicates the cost of running
spawned work on the local machine, which is the case that occurs most often for divide-
and-conquer problems. An overhead factor of 1.0 indicates that the divide-and-conquer
version executes as fast as its sequential counterpart.

The simplest program we ran is the Fibonacci example shown in Figure 5.1. This
benchmark is used by other divide-and-conquer systems as well, so we can use it to
compare Satin's sequential performance to other available systems. Fibonacci gives an
indication of the worst-case overhead, because it is very fine grained. A spawn is done
just for adding two integer numbers. Realistic applications are more coarse grained, and
thus have less overhead.

Cilk (version 5.3.1) is very efficient [62], the parallel Fibonacci program on one ma-
chine has an overhead of only a factor of 3.6 (on our DAS hardware). Atlas is imple-
mented completely in Java and does not use on-demand serialization. Therefore its over-
head is much worse, a factor of 61.5 is reported in [14]; the hardware used for the Atlas
measurements is not specified in [14]. The overhead of Satin is a factor of 7.25, sub-
stantially lower than that of Atlas. Satin is somewhat slower than Cilk, as Satin has to go
from Java to C and back, because the Satin runtime system is written in C. Besides the JNI
(Java Native Interface), Manta offers a special, more efficient native interface, which we
use for Satin. However, the overhead of calling C functions from Java is still significant.

Calling an empty virtual method costs about 0.07 μs in Manta. Figure 5.5 shows the
cost of spawn operations in Satin, for different numbers and types of parameters. The
numbers show that the overhead of a spawn operation is small, only 1.6 μs for a spawn
without parameters and without a return value. This means that Satin can spawn at most
625 thousand of these jobs per second on our hardware. As the table shows, the cost of a
spawn does not depend on the parameter type. All spawned method invocations with one
parameter take 1.80 μs. Adding a return value to a spawn adds less than 0.1 μs, regardless
of the type (not shown in Table 5.5). Adding parameters to the spawn operations increases

threshold	# spawns	run time (s)
10	$8.7 \cdot 10^8$	183.906
15	$7.8 \cdot 10^7$	159.415
20	$7.0 \cdot 10^6$	157.022
23	$1.7 \cdot 10^6$	156.863
25	$6.4 \cdot 10^5$	156.742
30	$5.7 \cdot 10^4$	157.001
35	$5.2 \cdot 10^3$	160.448
40	$4.7 \cdot 10^2$	188.029
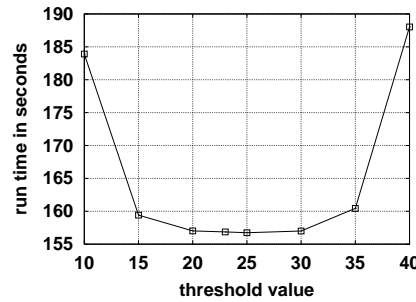
Figure 5.7: Fib performance on 64 machines, with different threshold values.

Figure 5.8: Fib run times on 64 machines, with different threshold values.

the cost linearly. The spawn cost increases with 0.16 μs per parameter on average, as is shown in Figure 5.6. The cost of a spawn operation is not dependent on the graph that can be reached via the parameters, due to the serialization-on-demand mechanism. No data is copied during a spawn operation. For example, a spawn with five object parameters and an object return value, takes about 2.5 μs. This means that Satin can still spawn 400 thousand of these jobs per second, independent of the complexity of the data structure that the objects point to.

The aforementioned overhead factors can be reduced at the application level by introducing threshold values, so that only large jobs are spawned. For Fibonacci, for example, we tried a threshold value of 23 for a problem of size 50, so all calls to *fib(n)* with $n<23$ are executed sequentially, without using a spawn operation. This simple change to the application reduced the overhead to almost zero. Still, 1.7 million jobs were spawned, leaving enough parallelism for running the program on large numbers of machines.

Our experience is that, due to Satin's low overhead, the tuning of the thresholds is not difficult. For Fibonacci for instance, we experimented with thresholds ranging from 10 to 40, resulting in a breakdown of the problem into between $8.7 \cdot 10^8$ and 465 jobs respectively. The number of spawned jobs for the different thresholds, and the resulting run times on 64 machines, are shown in Figure 5.7. The run times are also shown in Figure 5.8. The run times on 64 machines are used to investigate whether the application spawns enough work, as there must be an adequate amount of parallelism to balance the load of the computation on many machines.

The results show that with threshold values between 15 and 35, performance is within 2% of the shortest run time of Fibonacci (with the optimal threshold of 25). The difference between the number of jobs that is spawned in the low and high end of the good performing range (within 2% of the optimal threshold) is *four orders of magnitude*. We conclude that the low overhead of Satin makes it less important to fine-tune the threshold values. As long as the number of spawned jobs is sufficient to balance the load on the number of machines that is used, and within several orders of magnitude of the optimal value, the program is likely to run efficiently.

For Fibonacci, the threshold can easily be determined by the programmer, while for other applications this may be more difficult. However, for all applications we use in this

application	problem size	# spawns	t_s (s)	t_1 (s)	avg. thread length	overhd. factor
adaptive integration	0, 8E5, 1E-4	$1.9 \cdot 10^6$	4570.705	4466.759	2.326 ms	0.98
set covering problem	64, 32	$8.0 \cdot 10^6$	6480.704	6555.826	0.822 ms	1.01
fibonacci	44	$2.3 \cdot 10^9$	624.054	4918.297	2.167 μs	7.88
fibonacci threshold	50	$1.7 \cdot 10^6$	9924.717	10011.980	6.017 ms	1.00
iterative deepening A*	64	$1.6 \cdot 10^7$	6208.373	6344.855	0.377 ms	1.02
knapsack problem	30	$1.0 \cdot 10^6$	5172.664	5252.033	5.009 ms	1.01
matrix multiplication	1536 x 1536	$3.7 \cdot 10^4$	470.841	470.788	12.571 ms	1.00
n over k	36, 18	$1.3 \cdot 10^5$	4378.845	4648.738	35.467 ms	1.06
n-queens	20	$6.7 \cdot 10^4$	9977.964	9648.565	143.822 ms	0.97
prime factorization	9678904321	$2.1 \cdot 10^6$	7880.448	7623.762	3.635 ms	0.97
raytracer	balls2.nff	$3.5 \cdot 10^5$	8873.648	8878.225	25.401 ms	1.00
traveling sales person	19	$7.4 \cdot 10^5$	5877.088	6075.400	8.166 ms	1.03

Table 5.1: Application overhead factors.

chapter, we were able to find suitable threshold value without problems. In general, it is important to keep the sequential overhead of a divide-and-conquer system as small as possible, as it allows the creation of more fine-grained jobs and thus better load balancing.

The overhead for the other applications we implemented is much lower than for the (original) Fibonacci program, as shown in Table 5.1. Here, t_s denotes the run time of the sequential program, t_1 the run time of the parallel program on one machine. In general, the overhead depends on the number of parameters to spawned methods and the amount of work that is executed between spawn operations. All parameters have to be stored in the invocation record when the work is spawned, and pushed on the stack again, when executed. The overhead factor of Fibonacci in Table 5.1 is slightly larger than the afore-mentioned factor 7.25, because statistics are gathered during the run (e.g., the number of spawn and sync operations), causing a small extra overhead. For three applications, adaptive integration, n-queens and prime factorization, the parallel Satin program on one machine is actually reproducibly faster than the sequential program. We can only at-tribute this to caching effects, as the applications are deterministic, and the Satin program executes more code.

5.3.2   Parallel Application Performance

Communication in Satin is implemented by reusing serialization and communication code from the Manta RMI system (see Chapter 3). On the Myrinet network, the Satin round-trip time (a steal request and the reply message containing the job) for a spawned method invocation (or RMI) without parameters is 37 microseconds. The maximum throughput is 54 MByte/s, when the parameter to the spawned method invocation (or RMI) is a large array of a primitive type.

We ran twelve applications on the DAS cluster, using up to 64 machines. The ap-plications are described in Section 2.7. Figure 5.9 shows the achieved speedups while Table 5.2 provides detailed information about the parallel runs. All speedup values were computed relative to the sequential applications, compiled with the normal Manta com-
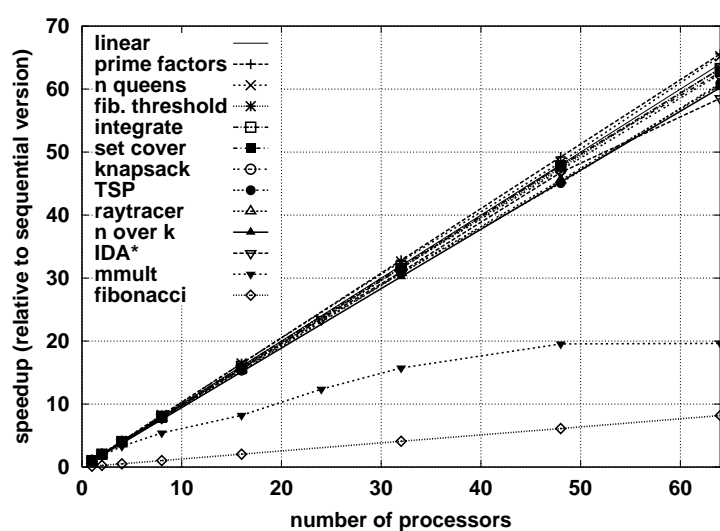
Figure 5.9: Application speedups.

application	time (s) 64 CPUs	speedup 64 CPUs	64 CPUs/ overhead	% max speedup	stolen	local/ stolen	MByte sent
integrate	72.265	63.25	65.49	96.6 %	7704	249.3	3.044
set cover	103.418	62.67	63.27	99.1 %	8252	967.0	26.174
fibonacci	76.293	8.18	8.12	100.7 %	11693	194116.7	2.842
fib thresh.	156.863	63.27	63.44	99.7 %	7902	210.6	2.825
IDA*	106.048	58.54	62.62	93.5 %	9324	1804.4	113.406
knapsack	82.907	62.39	63.03	99.0 %	9589	109.4	7.938
mat. mult.	23.988	19.63	64.04	30.7 %	15487	2.4	1421.306
n over k	72.700	60.23	60.28	99.9 %	4782	27.4	2.316
n-queens	153.007	65.21	66.18	98.5 %	4753	14.1	42.019
prime fact.	120.168	65.58	66.15	99.1 %	7586	276.5	2.244
raytracer	146.388	60.62	63.97	94.8 %	2349	148.8	39.615
TSP	96.483	60.91	61.91	98.4 %	4119	180.6	36.133

Table 5.2: Parallel performance breakdown for 64 machines.

piler (i.e., without the Satin extensions). A common viewpoint is that speedups should be relative to a sequential program that implements the *best known* algorithm, which may not be the divide-and-conquer version. However, we use thresholds for all applications to limit the parallelism and, when the threshold is reached, switch to a sequential algorithm to solve the remaining subproblem. We used the *best known* algorithm to sequentially solve the subproblems. The only exception is Fibonacci, for which an iterative linear time algorithm is known. Using this algorithm, the problem we use is computed within a millisecond. However, because we use Fibonacci mainly as a benchmark, the fact that there is a much better algorithm is not relevant. All other programs spend only a small fraction of the total run time in the splitting and combining of the subproblems. For the applications we used this is in the order of one percent at most.

One might think that setting the threshold to zero (thus computing everything with the sequential algorithm) would thus result in the best sequential algorithm, and we could use that to calculate the speedups. However, it turns out that this is not always the case. It is well known that divide-and-conquer algorithms have excellent cache behavior [63, 64, 146]. This means that the divide-and-conquer algorithms can sometimes perform better than non-recursive codes, especially when relatively large data sets are used (that do not fit in the cache). This happens because the divide-and-conquer algorithms keep splitting the problem. This often means that also the data set is split. At some point, the data set will fit in the cache. As most time is spent at the bottom of the tree, the algorithm mostly runs using data that can be kept in the cache, leading to good performance.

A good example of this is matrix multiplication. The program uses spawns to split the matrices until a certain threshold is reached (blocks of 48x48 in our version). Next, it multiplies the resulting blocks with an efficient sequential algorithm that steps 2 rows and columns at a time. The Satin program actually performs 3.5 times better than a program that only uses the sequential algorithm, due to more efficient use of the processor cache. The two blocks that are multiplied and the result all fit in the cache. The original sequential program linearly traverses the entire matrix, which does not fit in the cache.

We argue that it is valid to calculate the speedups relative to the Satin programs compiled with the normal Manta compiler (i.e., without using Satin's extentions), because, on the one hand, the programs spend only a small fraction of the total run time in the splitting and combining of the subproblems. On the other hand, due to cache effects, solving the problem partially with the divide-and-conquer algorithm and partially with the sequential algorithm, can lead to significant performance gains in some cases.

An interesting problem with writing the TSP application in Satin is that each parallel job needs to have access to the distance table (a two-dimensional array giving the distance between any two cities). Since Satin methods cannot access global data, this table is passed as a parameter in every recursive call. Due to Satin's on-demand serialization, however, the table is always passed around by reference (without copying or serialization) when the job is executed by the same machine that generated it. Only when the job actually gets stolen, the distance table is serialized and sent over the network. As can be seen in Table 5.2, only one out of 180 jobs is stolen. The overhead of the extra parameter thus is small.

Another possibility is to replicate the distance table with RepMI. However, as we will show in Section 5.3.3, reading data from a replicated object incurs some overhead. We

invocation	time *get* (μs)	time *set* (μs)
normal	0.08	0.11
replicated local	0.48	25.75
replicated 64 machines	0.48	120

Table 5.3: Overhead of replicated objects.

found that it is more efficient to pass the distance table as a parameter. Because Satin implements serialization-on-demand, the distance table is only serialized when a job is stolen. When a replicated object is used, the overhead of reading the replicated distance table is also paid for local work.

There is a strong correlation between measured speedup and the sequential overhead value, as already shown in Table 5.1: the lower the overhead, the higher the speedup we achieved. In Table 5.2 we compare the measured speedup with its upper bound, computed as the number of machines divided by the overhead on a single machine. We also show the percentage of this upper bound as actually achieved by the measured speedup. This percentage is very high for most applications, denoting that Satin's communication costs are low. The actual percentage depends (like the sequential overhead) on the number of method parameters and their total serialized size.

Table 5.2 also lists the total number of stolen jobs (Table 5.1 shows the total number of spawned jobs), and the ratio between the spawned and stolen jobs, which is less than 1 out of 14 for all applications, except for matrix multiplication. For most applications the ratio is much larger. Because the number of stolen jobs is so small, speedups are mainly determined by sequential overhead. A good example is Fibonacci, which achieves 100.7% of the upper bound, but still has a low speedup due to the sequential overhead. Satin's sequential efficiency thus is important for its successful deployment. Applications can achieve more that 100% of the upper bound, because 64 machines have 64 times as much cache and main memory as one machine. Table 5.2 also shows the total amount of data that is sent during the run, summed over all machines.

Matrix multiplication does not get good speedups, because the problem size is small due to memory constraints, the run time on 1 machine is only 470 seconds. With a perfect speedup, this would mean that the run time on 64 machines would be about 7 seconds. Also, much data is transferred, in total over all machines, as shown in Table 5.2, 1.4 GByte is sent during the run of 24 seconds (61 MByte/second).

5.3.3 Performance of Replicated Objects

We will now evaluate the performance of the replicated objects that RepMI provides in combination with Satin. We will use the TSP example from section 5.2.5 to show that replicated objects are a useful extension to the Satin programming model.

Low-level benchmarks

The advantage of using replicated objects instead of RMI is that read operations can be executed locally. Write operations are broadcast to all hosts where a replica is present. Thus,

TSP version	run time (s)	speedup	overhead (s)	
			replication	search
local deterministic	100.7	58.4	-	0
local nondeterministic	336.1	17.5	-	235.4
replicated deterministic	118.2	49.7	17.5	0
replicated nondeterministic	119.3	49.3	17.5	1.1

Table 5.4: performance of TSP on 64 machines with and without replicated objects.

replicated objects work well when the access pattern has a high read/write ratio. However, even local read operations are considerably more expensive than a normal method invocation. Table 5.3 shows the time for local invocations and invocations on the replicated *Minimum* object from Figure 5.4. A read operation on a replicated object (independent on the number of machines) takes $0.48\ \mu$s, six times more than a normal method invocation. Write operations on replicated objects are expensive, even on a single machine.

TSP With a Replicated Minimum Object

We investigated the performance of the TSP program with and without a replicated *Minimum* object (see Figure 5.4) on 64 machines. The results are shown in Table 5.4. The upperbound of the performance that can be achieved with TSP is the time for the deterministic version of TSP without the replicated minimum (labeled "local deterministic" in Table 5.4). The deterministic version uses a local minimum, which is initialized to the shortest path that is possible. Hence, all reads are local, and updates are never done. There is no search overhead, as all paths that are longer than the shortest route are immediately pruned. Of course, this version is "cheating", because it already uses the optimal answer, while the run still has to be done.

When the minimum is not initialized to the shortest path, the program is non deterministic. It uses a local minimum *per machine* to prune work. While this is more efficient than not pruning any work at all, considerable search overhead is still present. When no work is pruned, TSP takes several hours. With the local minimum values per machine, this is reduced to 336 seconds (see Table 5.4).

With a replicated *Minimum* object, search overhead can be even further reduced. However, as shown in Table 5.3, a local read operation of the replicated minimum value is six times more expensive than a normal method invocation. We investigated the impact of the slower read operation by using the program with the replicated minimum, but with the value of the minimum set to the shortest path (labeled "replicated deterministic" in Table 5.4. The resulting program is thus deterministic, and can be compared with the deterministic program without the replicated object. As can be seen in Table 5.4, the overhead of the replicated minimum is significant. The deterministic program with the replicated object is 17% slower than the deterministic version without the replicated minimum.

However, the measurements in Table 5.4 show that the replicated object reduces the search overhead to a minimum: the nondeterministic version without the pre-initialized minimum value (labeled "replicated nondeterministic") is only one second slower than

the deterministic program with the replicated object (i.e., the program without search overhead). The nondeterministic program quickly finds a minimal path, and only 9 minimum updates are needed. The rest of the work can be effectively pruned when the route becomes longer than the minimum value. Even though the replicated objects do have a considerable overhead, the reduction in search overhead makes it worthwhile to use them. When the minimum is not initialized to the shortest path, the version with the replicated object is 2.8 times faster than the version that uses a minimum per machine.

5.4   Related Work

We discussed Satin, a divide-and-conquer extension of Java. Satin is designed for wide-area systems, without shared memory. Many divide-and-conquer systems are based on the C language. Among them, Cilk [22] only supports shared-memory machines, Cilk-NOW [24] and DCPAR [61] run on local-area, distributed-memory systems, but do not support shared data. Also, they do not make a deep copy of the parameters to spawned methods. SilkRoad [129] is a version of Cilk for distributed memory systems that uses a software DSM to provide shared memory to the programmer, targeting at small-scale, local-area systems. Alice [42] and Flagship [169] offer specific hardware solutions for parallel divide-and-conquer programs (i.e., a reduction machine with one global address space for the parallel evaluation of declarative languages). Satin is purely software based, and neither requires nor provides a single address space. Instead, it uses Manta's object-replication mechanism to provide shared objects [111].

Blumofe et al. [22] describe a way to calculate the available average parallelism in divide-and-conquer applications. The amount of parallelism is defined as the work divide by the critical path length of the application. The work is the time it takes to run the parallel program on a single machine, while the critical path is the time it takes to run the program on an infinite number of machines. This is equivalent to the largest sum of thread execution times along any path. The average parallelism is a theoretical upperbound on the scalability of the application, and can be used, for instance, to tune the threshold values. However, we found that, in practice, it is not a problem to tune the thresholds. Moreover, the performance model in [22] does not take communication costs into account. Therefore, the amount of parallelism, when calculated as described above, can considerably overestimate the real scalability of the application.

An interesting (hierarchical) system is HyperM [166], which was used to run parallel divide-and-conquer applications on a distributed memory machine. Here, a single primitive, called the sandwich annotation, was used to indicate parallelism in a functional language.

Mohr et al. [116] describe the importance of avoiding thread creation in the common, local case (lazy task creation). Targeting distributed memory adds the problem of copying the parameters (marshalling). Satin builds on the ideas of lazy task creation, and avoids both the starting of threads and the copying of parameter data by choosing a suitable parameter passing mechanism.

Another divide-and-conquer system based on Java is Atlas [14]. Atlas is not a Java extension, but a set of Java classes that can be used to write divide-and-conquer programs.

While Satin is targeted at efficiency, Atlas was designed with heterogeneity and fault tolerance in mind, and aims only at a reasonable performance. Because Satin is compiler based, it is possible to generate code to create the invocation records, thus avoiding all runtime type inspection. The Java classes presented by Lea et al. ([100]) can also be used for divide-and-conquer algorithms. However, they are restricted to shared-memory systems.

A compiler-based approach is also taken by Javar [20]. In this system, the programmer uses annotations to indicate divide-and-conquer and other forms of parallelism. The compiler then generates multithreaded Java code, that runs on any JVM. Therefore, Javar programs run only on shared-memory machines and DSM systems, whereas Satin programs run on wide-area systems with distributed memory. Java threads impose a large overhead, which is why Satin does not use threads at all, but uses lightweight invocation records instead.

Herrmann et al. [74] describe a compiler-based approach to divide-and-conquer programming that uses skeletons. The DHC compiler supports a purely functional subset of Haskell [82], and translates source programs into C and MPI.

5.5   Conclusion

We have described our experiences in building a parallel divide-and-conquer system for Java, which runs on distributed memory machines. The programming model uses spawn and sync operations to express parallelism. We have shown that an efficient implementation is possible by choosing convenient parameter semantics. An important optimization is the on-demand serialization of parameters to spawned method invocations. This was implemented using invocation records. Our Java compiler generates code to create these invocation records for each spawned method invocation. We have also demonstrated that divide-and-conquer programming can be cleanly integrated into Java, and that problems introduced by this integration (e.g., through garbage collection) can be solved. The results show that Satin programs can be efficiently executed in parallel on a cluster of workstations. We have shown that, although Satin's programming model only supports the spawning of methods without side-effects, other communication systems (e.g., RepMI) can be used to effectively circumvent this restriction. In the next chapter, we will show how Satin applications can be executed efficiently on hierarchical wide-area systems, without special optimizations by the application programmer.

Chapter 6

Load Balancing Wide-Area Divide-and-Conquer Applications

```
The one who adapts his policy to the times prospers,
and, likewise, the one whose policy clashes with
the demands of the times does not.

- Niccolo Machiavelli
```

In distributed supercomputing, platforms are often hierarchically structured. Typically, multiple supercomputers or clusters of workstations are connected via wide-area links, forming systems with a two-level communication hierarchy. When running parallel applications on such multi-cluster systems, efficient execution can only be achieved when the hierarchical structure is carefully taken into account. In Chapter 4, we have demonstrated that various kinds of parallel applications can indeed be efficiently executed on wide-area systems. However, each application had to be optimized individually to reduce the utilization of the scarce wide-area bandwidth or to hide the large wide-area latency. Some of these optimizations can be extracted into specific runtime systems like MPI's collective communication operations (the MagPIe library [89]), and our Java-based object replication mechanism RepMI [111]. In general, however, it is hard for a programmer to manually optimize parallel applications for hierarchical systems.

The ultimate goal of our work is to create a programming environment in which parallel applications for hierarchical systems are easy to implement. Ideally, the application programmer should not have to implement different wide-area optimizations for each application by hand. We chose one specific class of problems, divide-and-conquer algorithms, and implemented an efficient compiler and runtime system that apply wide-area optimizations automatically. The divide-and-conquer model lends itself well for hierarchically structured systems because tasks are created by recursive subdivision. This leads to a hierarchically structured task graph which can be executed with excellent communi-

cation locality, especially on hierarchical platforms.

Divide-and-conquer programs are easily parallelized by letting the programmer annotate potential parallelism in the form of *spawn* and *sync* constructs. To achieve efficient program execution, the generated work load has to be balanced evenly among the available CPUs. For single cluster systems, *Random Stealing* (RS) is known to achieve optimal load balancing. However, RS is inefficient when applied to hierarchical wide-area systems where multiple clusters are connected via wide-area networks (WANs) with high latency and low bandwidth.

In this chapter we study wide-area load-balancing algorithms for divide-and-conquer applications, using our Satin system. Satin's compiler and runtime system cooperate to implement the divide-and-conquer primitives efficiently on a hierarchical wide-area system, without requiring any help or optimizations from the programmer. Of course, load balancing is not the only issue that is important for running divide-and-conquer programs on the grid. Security, fault tolerance, etc. should also be dealt with. However, in this thesis, we focus on load balancing of divide-and-conquer applications.

Five load-balancing algorithms were implemented in the Satin runtime system to investigate their behavior on hierarchical wide-area systems. We will demonstrate that *Random Stealing* (RS), as used in single-cluster environments, does not perform well on wide-area systems. We found that *hierarchical load balancing*, as proposed for example by Atlas [14], Javelin 3 [120] and Dynasty [8], performs even worse for fine-grained applications. The hierarchical algorithm sends few wide-area messages, but suffers from bad performance inside clusters, and stalls the entire cluster when wide-area steals are issued. We introduce a novel algorithm, *Cluster-aware Random Stealing* (CRS), and show that it achieves good speedups for a large range of bandwidths and latencies, although it sends more messages than the other cluster-aware load-balancing methods. We compare CRS with *Random Stealing* and *hierarchical load balancing*, and two other algorithms, *Random Pushing* and *Cluster-aware Load-based Stealing*, a variant of hierarchical stealing. Both approaches are candidates for efficient execution on multi-cluster systems. We show, however, that CRS outperforms the other algorithms in almost all test cases.

We discuss the performance of all five load-balancing algorithms using twelve applications on the DAS system. We also describe the impact of different WAN bandwidth and latency parameters. We show that on a system with four clusters, even with a one-way WAN latency of 100 milliseconds and a bandwidth of only 100 KBytes/s, 11 out of 12 applications run only at most 4% slower than on a single cluster with the same total number of CPUs.

We investigate the performance of CRS further with a bandwidth-latency analysis and a scalability analysis. The results show that CRS can tolerate extreme WAN latencies. Even with WAN latencies of one second, the overhead compared to a single, local cluster is only 13% when CRS is used to balance the load of the computations. CRS is more sensitive to bandwidth, but the most bandwidth-sensitive application used in this chapter, a raytracer, still achieves a good speedup of 50.8 using 64 machines, with a WAN bandwidth of only 20 KByte/s. Furthermore, we show that CRS scales to many small clusters. Even with 16 clusters of only two nodes each, the run time is only increased by 5% compared to a single cluster.

Finally, we present a case study where we run Satin applications using more realistic grid scenarios. We investigate the performance of RS, CHS and CRS on non-fully connected systems, and with dynamically changing WAN link parameters. We also use a scenario that replays Network Weather Service (NWS) data of real wide-area links. CRS outperforms the other algorithms, but still performs sub-optimally in a few test cases where some WAN links are fast and some are very slow. We introduce another novel algorithm, *Adaptive Cluster-aware Random Stealing* (ACRS), which solves this remaining problem. The speedup of the four Satin applications we investigated in the case study all achieve a speedup above 59.0 on 64 machines with ACRS on the scenario that replays real NWS data. These strong results suggest that divide-and-conquer parallelism is a useful model for writing distributed supercomputing applications on hierarchical wide-area systems.

The contributions we make in this chapter are the following:

- we show that load-balancing algorithms that are currently in use in single-cluster systems (e.g., RS, RP) are not adequate in a grid environment;

- we demonstrate that hierarchical load-balancing algorithms, which are proposed in the literature for grid systems, and are already used by several projects (e.g., Atlas, Javelin), perform even worse for fine-grained applications;

- we describe a novel load-balancing algorithm (CRS), that performs extremely well on wide-area systems, even with high WAN latencies and low bandwidths;

- we present a bandwidth-latency analysis, and a scalability analysis for CRS to investigate what the performance characteristics of CRS are;

- in a case study, we run Satin applications using more realistic grid scenarios, for instance by replaying NWS data, and show that CRS outperforms previous load-balancing algorithms;

- we introduce another novel algorithm (ACRS) that performs even better than CRS in some extreme asymmetric scenarios that combine very fast and very slow wide-area links.

The remainder of the chapter is structured as follows. In Section 6.1 we present the five load-balancing algorithms. In Section 6.2 we evaluate their performance with twelve example applications. We investigate the performance of CRS further with a bandwidth-latency analysis in Section 6.3 and a scalability analysis in Section 6.4. We present a case study of some Satin applications with more realistic grid scenarios in Section 6.5. Related work on load balancing is discussed in Section 6.6. Finally, Section 6.7 concludes this chapter.

6.1 Load Balancing in Wide-Area Systems

To minimize application completion times, load-balancing algorithms try to keep all processors busy performing application-related work. The distribution of jobs causes com-

name	optimization goal	drawbacks	heuristics	work transfer
RS	communication	high idle time due to sync. WAN communication	no	synchronous
RP	idle time	unstable, too much WAN communication	yes	asynchronous
CHS	WAN communication	too much LAN traffic, cluster stalling	no	synchronous
CLS	LAN communication WAN latency hiding	slow work distribution inside cluster, prefetching bottleneck	yes	LAN synchronous WAN prefetching
CRS	LAN communication WAN latency hiding		no	LAN synchronous WAN asynchronous

Table 6.1: Properties of the implemented load-balancing algorithms.

munication among the processors that has to be performed in addition to the proper application work. Minimizing both the idle time and communication overhead are conflicting goals. Load-balancing algorithms have to carefully balance communication-related overhead and processor idle time [152]. In the case of multi-cluster wide-area systems, the differences in communication costs between the local-area (LAN) and wide-area networks (WAN) have to be taken into account as well, adding further complexity to the optimization problem.

In this section, we discuss the five different load-balancing algorithms we implemented in the Satin system. The properties of the algorithms are summarized in Table 6.1. For each algorithm, the table shows the optimization goal, the drawbacks it has, whether is uses heuristics or not, and the communication method (synchronous or asynchronous). We start with existing algorithms for single-cluster systems that try to optimize either communication overhead (*random stealing* (RS) [23]) or idle time (*random pushing* (RP) [152]). We show that both are inefficient for multi-cluster wide-area systems. Next, we investigate *Cluster-aware Hierarchical Stealing* (CHS) [8, 14, 120] which is believed to be efficient for hierarchical systems. However, by optimizing wide-area communication only, this approach leads to mediocre results due to excessive local communication and causes whole clusters to stall while waiting for remote work.

To overcome the drawbacks of the three existing algorithms, we implemented two new load-balancing algorithms. The first one, *Cluster-aware Load-based Stealing* (CLS), directly improves CHS. It combines RS inside clusters with work prefetching between clusters, the latter based on monitoring the load of all nodes. This algorithm performs quite well, but still suffers from two drawbacks: it relies on manually tuned parameters and propagates work rather slowly between clusters.

The second new algorithm is called *Cluster-aware Random Stealing* (CRS). It combines RS inside clusters with controlled asynchronous work stealing from other clusters. CRS minimizes LAN communication while avoiding high idle time due to synchronous WAN stealing. CRS does not need parameter tuning and is almost trivial to implement. In Section 6.2 we will show that, with CRS, 11 out of 12 applications running on multiple clusters are only marginally slower than on a single large cluster with the same number of processors, using RS.

All algorithms use a double-ended work queue on each node, containing the *invocation records* of the jobs that were spawned but not yet executed. Divide-and-conquer programs progress by splitting their work into smaller pieces, and by executing the small jobs that are not worth further splitting. New jobs are inserted at the head of the queue. Each processor fetches work from the head of its own queue. All load-balancing algorithms described here use the jobs at the tail of the queue for balancing the work load. This scheme ensures that the most fine-grained jobs run locally, while the jobs at the tail of the queue are the most coarse grained ones available. They are ideal candidates for load-balancing purposes.

6.1.1 Random Stealing (RS)

Random stealing is a well known load-balancing algorithm, used both in shared-memory and distributed-memory systems. RS attempts to steal a job from a randomly selected peer when a processor finds its own work queue empty, repeating steal attempts until it succeeds. This approach minimizes communication overhead at the expense of idle time. No communication is performed until a node becomes idle, but then it has to wait for a new job to arrive. RS is provably efficient in terms of time, space, and communication for the class of fully strict computations [23, 174]; divide-and-conquer algorithms belong to this class. An advantage of RS is that the algorithm is *stable* [152]: communication is only initiated when nodes are idle. When the system load is high, no communication is needed, causing the system to behave well under high loads.

On a single-cluster system, RS is, as expected, the best performing load-balancing algorithm. On wide-area systems, however, this is not the case. With N nodes participating in the computation, the chance that a steal attempt from a node inside a cluster of size CS will go to a remote cluster is $(N - CS)/(N - 1) * 100\%$. Since the steal attempts are synchronous (see Table 6.1), the stealing processor (the "thief") has to wait a wide-area round-trip time for a result, while there may be work available inside the local cluster. With four clusters of 16 machines each, this already affects 76% of all steal requests. When more clusters are used, this becomes even higher. Also, when jobs contain much data, the limited wide-area bandwidth becomes a bottleneck.

6.1.2 Random Pushing (RP)

Random pushing is another well known load-balancing algorithm [152]. With RP, a processor checks, after insertion of a job, whether the queue length exceeds a certain threshold value. If this is the case, a job from the queue's tail (where the largest jobs are) is pushed to a randomly chosen peer processor. This approach aims at minimizing processor idle time, because jobs are pushed ahead of time, before they are actually needed. However, this comes at the expense of additional communication overhead. One might expect RP to work well in a wide-area setting, because its communication is asynchronous (see Table 6.1) and thus less sensitive to high wide-area round-trip times than work stealing. A problem with RP, however, is that the algorithm is not *stable*. Under high work loads, job pushing causes useless overhead, because all nodes already have work. In fact, the higher the load is, the more communication overhead the algorithm has. Overloaded

machines try to push excess work to other highly loaded machines, which in turn have to distribute the work further.

Also, RP has the same problem as RS with respect to communication to remote clusters. With N nodes participating in the computation, work pushed from a node in a cluster of size CS will go to a remote cluster with a chance of $(N - CS)/(N - 1) * 100\%$, causing wide-area bandwidth problems. Unlike random stealing, random pushing does not adapt its WAN utilization to bandwidth and latency as it lacks a bound for the number of messages that may be sent, i.e., there is no inherent flow-control mechanism. Memory space is also not bounded: jobs may be pushed away as fast as they can be generated, and have to be stored at the receiver. To avoid exceeding of communication buffers, Satin's implementation of RP adds an upper limit of jobs sent by each node that can be in transit simultaneously. This upper limit has to be optimized manually. Additionally, a threshold value must be found that specifies when jobs will be pushed away. A single threshold value is not likely to be optimal for all applications, or not even for one application with different wide-area bandwidths and latencies. Simply pushing away all generated jobs is found to perform well in theory [152] (without taking bandwidth and latency into account), but, in practice, has too much communication and marshalling overhead for fine-grained divide-and-conquer applications.

6.1.3   Cluster-aware Hierarchical Stealing (CHS)

Random stealing and random pushing both suffer from too much WAN communication. Cluster-aware Hierarchical Stealing (CHS) has been presented for load balancing divide-and-conquer applications in wide-area systems (e.g., for Atlas [14], Javelin 3 [120] and Dynasty [8]). The goal of CHS is to minimize wide-area communication. The idea is to arrange processors in a tree topology, and to send steal messages along the edges of the tree. When a node is idle, it first asks its child nodes for work. If the children are also idle, steal messages will recursively descend the tree. Only when the entire subtree is idle, messages will be sent upwards in the tree, asking parent nodes for work.

This scheme exhibits much locality, as work inside a subtree will always be completely finished before load-balancing messages are sent to the parent of the subtree. By arranging the nodes inside a cluster in a tree shape, the algorithm's locality can be used to minimize wide-area communication. Multiple cluster trees are interconnected at their root nodes via wide-area links. When the cluster root node finds its own cluster to be idle, it sends a steal message to the root node of another, randomly selected cluster. Such an arrangement is shown in Figure 6.1.

CHS has two drawbacks. First, the root node of a cluster waits until the entire cluster becomes idle before starting wide-area steal attempts. During the round-trip time of the steal message, the entire cluster remains idle. Second, the preference for stealing further down the tree results in jobs with finer granularity to be stolen first, leading to high LAN communication overhead, due to many job transfers.

The actual implementation of hierarchical stealing is more complex. When a job is finished that runs on a different machine than the one it was spawned on, the return value must be sent back to the node that generated the job. While the result arrives, this node may be stealing from its children or parent. However, the incoming return value may
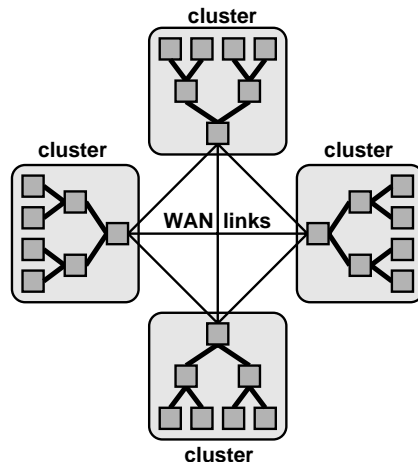
Figure 6.1: Cluster Hierarchical Stealing: arrange the nodes in tree shapes and connect multiple trees via wide-area links.

allow new jobs to be generated, so the receiver may not be idle anymore. In this case, Satin lets the receiver cancel its potentially very expensive steal request that might be forwarded recursively, even across wide-area links. However, we found that the benifit gained from this optimization is marginal in practice.

6.1.4   Cluster-aware Load-based Stealing (CLS)

CHS suffers from high LAN communication overhead, caused by stealing fine-grained jobs, and from the stalling of a whole cluster when stealing across a wide-area connection. In order to address these two problems, while minimizing the wide-area communication, we developed Cluster-aware Load-based Stealing (CLS). The idea behind CLS is to combine random stealing inside clusters with wide-area work prefetching performed by one coordinator node per cluster. With a single cluster, CLS is identical to RS.

Random stealing inside a cluster does not have the preference for stealing fine-grained jobs and thus reduces the LAN communication overhead, compared to the tree-based approach. The WAN communication can be controlled and minimized by letting only coordinator nodes perform wide-area work stealing. To avoid whole-cluster stalling, the coordinator nodes can prefetch jobs. Prefetching requires a careful balance between communication overhead and processor idle time. On the one hand, when jobs are prefetched too early, the communication overhead grows unnecessarily. On the other hand, when jobs are prefetched too late, processors may become idle. Pure work stealing can be seen as one extreme of this tradeoff, where jobs are never prefetched. Pure work pushing is the other extreme where jobs are always transferred ahead of time.

In CLS, prefetching is controlled by load information from the compute nodes. Each node periodically sends its load information to the cluster coordinator that monitors the

overall load of its cluster. The compute nodes send their load messages asynchronously, keeping the overhead small. Satin further reduces this overhead by sending periodical load messages only when the load actually has changed. Furthermore, when a node becomes idle, it immediately sends a load message (with the value zero) to the coordinator. A good interval for sending the load messages is subject to parameter tuning. It was empirically found to be 10 milliseconds. This reflects the rather fine granularity of Satin jobs. Sending load messages with this interval does not noticeably decrease the performance. When the total cluster load drops below a specified threshold value, the coordinator initiates inter-cluster steal attempts to randomly chosen nodes in remote clusters. The coordinator can hide the possibly high wide-area round-trip time by overlapping communication and computation, because wide-area prefetch messages are sent asynchronously. The coordinator also performs local steals concurrently with the wide-area steal attempts.

Another tunable parameter is the threshold value that is used to trigger wide-area prefetching. This parameter strongly depends on the application granularity and the WAN latency and bandwidth. In Satin's implementation of CLS, we use the length of the work queue as load indicator. This indicator might not be very accurate, due to the job granularity that descends with increasing recursion depth. We found that using the recursion depth as measure of a job's size does not improve performance. The DCPAR system [61] uses programmer annotations to express job granularities, but even this approach fails with irregular applications that perform pruning in the task tree. Despite its weakness, using the queue length as load indicator is our only choice, as additional, accurate information is unavailable. Our results confirm those of Kunz [94], who found that the choice of load indicator can have a considerable impact on performance, and that the most effective one is queue length. Furthermore, he found no performance improvements when combinations of indexes were used. Kunz obtained his results in the context of scheduling Unix processes on a cluster of workstations.

The amount of prefetching (and thus the amount of wide-area communication) may be adjusted by tuning the load threshold for wide-area stealing. A value of '1' initiates inter-cluster communication only when there are no jobs left in the local cluster. This minimizes the number of wide area messages, at the cost of load imbalance. This compares to hierarchical stealing, which also waits until the entire cluster becomes idle, before sending wide-area steal messages. One important difference is that with hierarchical stealing, the entire cluster is traversed sequentially, while in the load-based approach, the idle nodes immediately contact their coordinator node.

6.1.5   Cluster-aware Random Stealing (CRS)

The CLS algorithm described above minimizes LAN communication overhead while simultaneously reducing WAN communication and idle time. However, CLS relies on careful parameter tuning for the volume of job prefetching. Furthermore, prefetched jobs are stored on a centralized coordinator node rather than on the idle nodes themselves. The distribution of jobs to their final destination (via random stealing) adds some overhead to this scheme. With high wide-area round-trip times, the coordinator node might even become a stealing bottleneck if the local nodes together compute jobs faster than they can be prefetched by the coordinator node.

We designed Cluster-aware Random Stealing (CRS) to overcome these problems. Like CLS, it uses random stealing inside clusters. However, it uses a different approach to wide-area stealing. The idea is to omit centralized coordinator nodes at all. Instead, we implement a decentralized control mechanism for the wide-area communication directly in the worker nodes. Pseudocode for the new algorithm is shown in Figure 6.2. In CRS, each node can directly steal jobs from nodes in remote clusters, but at most one job at a time. Whenever a node becomes idle, it first attempts to steal from a node in a remote cluster. This wide-area steal request is sent asynchronously. Instead of waiting for the result, the thief simply sets a flag and performs additional, synchronous steal requests to nodes within its own cluster, until it finds a new job. When local work is found, the wide-area steal request is not canceled. As long as the flag is set, only local stealing will be performed. The handler routine for the wide-area reply simply resets the flag and, if the request was successful, puts the new job into the work queue.

The remote victim is chosen by repeatedly selecting a random host out of the total pool of hosts, until a machine in a *remote* cluster has been found. This way, all remote nodes have the same chance of being chosen. Alternatively, it would be possible to first select a remote cluster at random, and then a node inside that cluster. The latter scheme, however, does not work well when the clusters are not of equal size, because nodes in small clusters have a larger chance of being selected than nodes in larger clusters. This is undesirable, as nodes in small clusters will be drained of work.

As will be shown in Section 6.2, this asynchronous wide-area stealing successfully hides the long wide-area round-trip times. The mechanism also implements an efficient way of job prefetching that delivers the new job directly on the idle node and does not need parameter tuning. The implication of this scheme is that many (or all) remote clusters will be asked for work concurrently when a large part of a cluster is idle. As soon as one remote steal attempt is successful, the work will be quickly distributed over the whole cluster, because local steal attempts are performed during the wide-area round-trip time. Thus, when jobs are found in a remote cluster, the work is propagated quickly.

Compared to RS, CRS significantly reduces the number of messages sent across the wide-area network. CRS has the advantages of random stealing, but hides the wide-area round-trip time by additional, local stealing. The first job to arrive will be executed. No extra load messages are needed, and no parameters have to be tuned. On a single cluster, CRS is identical to RS. Still, the CRS algorithm is *stable* [152]: communication is only initiated when nodes are idle. This applies to both local-area and wide-area communication. When the system load is high, no new messages are sent, the algorithm thus behaves well under high loads.

6.1.6 Alternative Algorithms

There may be many more load-balancing algorithms that could perform well in a wide-area setting. For example, Eager et al. [47] describe a form of work pushing, where the sender first polls target nodes, until one is found that is not overloaded. Only after this node is found, the work is transferred. In a wide-area setting, this would imply that multiple wide-area round-trip times may be needed to transfer work, hence giving up the benefit of asynchronous WAN communication of our RP algorithm. However, for

```
1  int remote_victim(void) {
2      int dest;
3      do {
4          dest = random(nr_total_hosts);
5      } while (cluster_of(dest) == my_cluster);
6      return dest;
7  }
8
9  void cluster_aware_random_stealing(void) {
10    while(!exiting) {
11      job = queue_get_from_head();
12      if(job) {
13        execute(job);
14      } else {
15        if(nr_clusters > 1 && !stealing_remotely) {
16          /* no wide-area message in transit */
17          stealing_remotely = true;
18          send_async_steal_request(remote_victim());
19        }
20        /* do a synchronous steal in my cluster */
21        job = send_steal_request(local_victim());
22        if(job) queue_add_to_tail(job);
23      }
24    }
25  }
26
27  void handle_wide_area_reply(Job job) {
28    if(job) queue_add_to_tail(job);
29    stealing_remotely = false;
30  }
```

Figure 6.2: Pseudo code for Cluster-aware Random Stealing.

bandwidth sensitive applications, this may perform better than the form of work pushing implemented by Satin, because jobs are not sent over wide-area links unless they are needed in the remote cluster.

Also, some of Satin's algorithms could be further improved. For example, our CLS implementation selects a random stealing victim in a random cluster. It could also ask the remote cluster coordinator what the best node in the cluster is. It is unclear what the performance of this scheme would be, because load information is outdated quickly. The hierarchical scheme, CHS, might be tuned by trying different tree shapes inside the clusters. The algorithms that are sensitive to parameter tuning (RP and CLS) might be improved by adaptively changing the threshold values at run time.

We do not claim that we have investigated all possibilities, but we presented a number of load-balancing algorithms that span the spectrum of possible design alternatives. We described a simple, cluster-aware algorithm (CRS) that performs almost as good on hierarchical wide-area systems as random stealing does in a single cluster (performance results will be shown in the next section). CRS does not rely on parameter tuning, but adapts itself to given WAN parameters. There may be other algorithms that perform

equally well, but they are probably more complicated. Therefore, we argue that CRS is well-suited for load balancing divide-and-conquer applications on hierarchical wide-area systems.

6.2   Performance Evaluation

We have evaluated Satin's performance using 12 application kernels, which are described in Section 2.7. The problem sizes used are the same as in Chapter 5 (see Table 5.1). None of the applications described in this chapter use Manta's replicated object system (RepMI [111]), allowing us to focus on the communication patterns of the load-balancing algorithms. Manta's replicated objects work well in wide-area systems, because a special, cluster-aware broadcast is used to send updates to the replicas. A more detailed performance analysis of RepMI on wide-area systems is given in [111].

Satin's load-balancing algorithms are implemented on top of the Panda communication library [12], which has efficient implementations on a variety of networks. On the Myrinet network (which we use for the measurements in this chapter), Panda is implemented on top of the LFC [19] network interface protocol. Between clusters, Panda uses TCP. Satin uses Panda's efficient user-level locks for protecting the work queue. To avoid the overhead of operating-system calls, both LFC and Panda run in user space.

A major challenge in investigating the performance of the Satin applications is the actual WAN behavior. Typical wide-area links are part of the Internet and thus shared among many applications, making run time measurements irreproducible and thus scientifically hardly valuable. To overcome this problem, we use the WAN emulator that is a part of Panda. The WAN emulator allows us to run parallel applications on a single (large) parallel machine with only the wide-area links being emulated. The Panda emulator is highly accurate and configurable at run time.

All measurements were performed on the cluster of the Distributed ASCI Supercomputer (DAS) at the Vrije Universiteit (see Section 1.9). Our results have been obtained on a *real parallel machine*; only the wide-area links are simulated by letting the communication subsystem insert delay loops into message delivery [132]. This allows us to investigate the performance of the load-balancing algorithms with different communication performance parameters. We have verified some of the results on four geographically distributed Myrinet clusters of the DAS, which are connected by *real* wide-area links. The results on this system are consistent with the results we provide here, with simulated wide-area links.

6.2.1   The Panda Wide-area Network Emulator

Satin's load-balancing algorithms are implemented on top of the Panda communication library. Panda allows us to run parallel applications across multiple (DAS) clusters. For this purpose, one dedicated node in each cluster acts as a *gateway*. Whenever an application node wants to send a message to a node in a different cluster, it sends the message to its local gateway node, which in turn forwards it to the gateway node of the remote cluster, where the message gets forwarded to the receiver node. Between cluster gateways, Panda
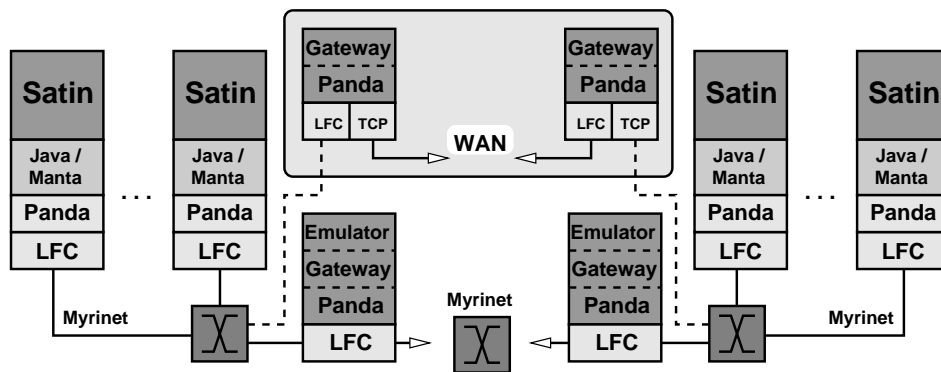
Figure 6.3: Local and wide-area communication with Panda and the WAN emulator.

communicates using the standard TCP protocol. This communication path is shown in Fig. 6.3, using the upper, shaded path between the two clusters (on the left and on the right sides).

The WAN emulator allows us to run parallel applications on a single (large) cluster with only the wide-area links being emulated. For this purpose, Panda provides an emulator version of its gateway functionality. Here, communication between gateway nodes physically occurs inside a single cluster, in our case using Myrinet. This communication path is shown in Fig. 6.3, using the lower path between the two clusters.

The actual emulation of WAN behavior occurs in the receiving cluster gateways which delay incoming messages before forwarding them to the respective receivers. On arrival of a message from a remote cluster, the gateway computes the emulated arrival time, taking into account the emulated latency and bandwidth from sending to receiving cluster, and the message length. The message is then put into a queue and gets delivered as soon as the delay expires. The latency and bandwidth can be specified for each individual link. With this setup, the WAN emulation is completely transparent to the application processes, allowing realistic and at the same time reproducible wide-area experimentation.

We also investigated the precision of our emulator. Therefore, we measured bandwidth and latency between the DAS clusters using ping-pong tests with messages of varying sizes. We then fed the measured parameters into the emulator and re-ran our tests. Fig. 6.4 compares real and emulated latency and bandwidth between the DAS clusters at the VU (Amsterdam) and Delft University of Technology (in both directions, so there are four lines in each graph). In the graphs, the respective pairs of lines are hardly distinguishable, giving evidence for the close match between the real system and its emulation. The measurements for the other wide-area DAS links show similar behavior.

6.2.2 Wide-area Measurements

Our measurements show that the total send overhead, such as parameter marshalling and buffer copying, is at most 1% of the run time for all applications and load-balancing
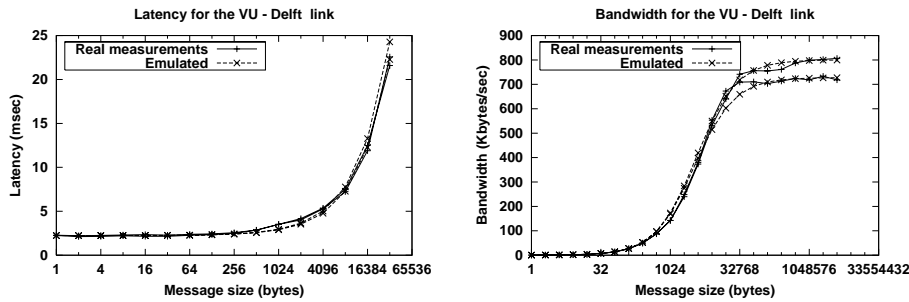
Figure 6.4: Measured vs. emulated latency and bandwidth between 2 DAS clusters (in both directions).

algorithms. The speedups of the twelve applications, relative to the sequential program (without spawn and sync), are shown in Figures 6.5 and 6.6, for all five load-balancing algorithms we described. The first (white) bar in each graph is the speedup on a single cluster of 64 nodes (i.e., without wide-area links). The following four bars indicate runs on four clusters of 16 nodes each, with different wide-area bandwidths and latencies. The results for RP contain a number of small anomalies, where speedups increase as wide-area speed is decreased. These anomalies are due to the nondeterministic behavior of the algorithm.

Matrix multiplication is the only application that does not perform well. As explained in Section 5.3.2, we cannot make the problem large enough to obtain good performance, due to memory constraints. On four clusters, CLS performs best for matrix multiplication, but even then, the speedup is only 11 with a one-way WAN latency of 10 milliseconds and a throughput of 1000 KByte/s. On one cluster of 16 nodes, the speedup is 9.02, so adding three extra clusters only slightly improves performance. Therefore, we conclude that the divide-and-conquer version of Matrix multiplication is not well suited for wide-area systems.
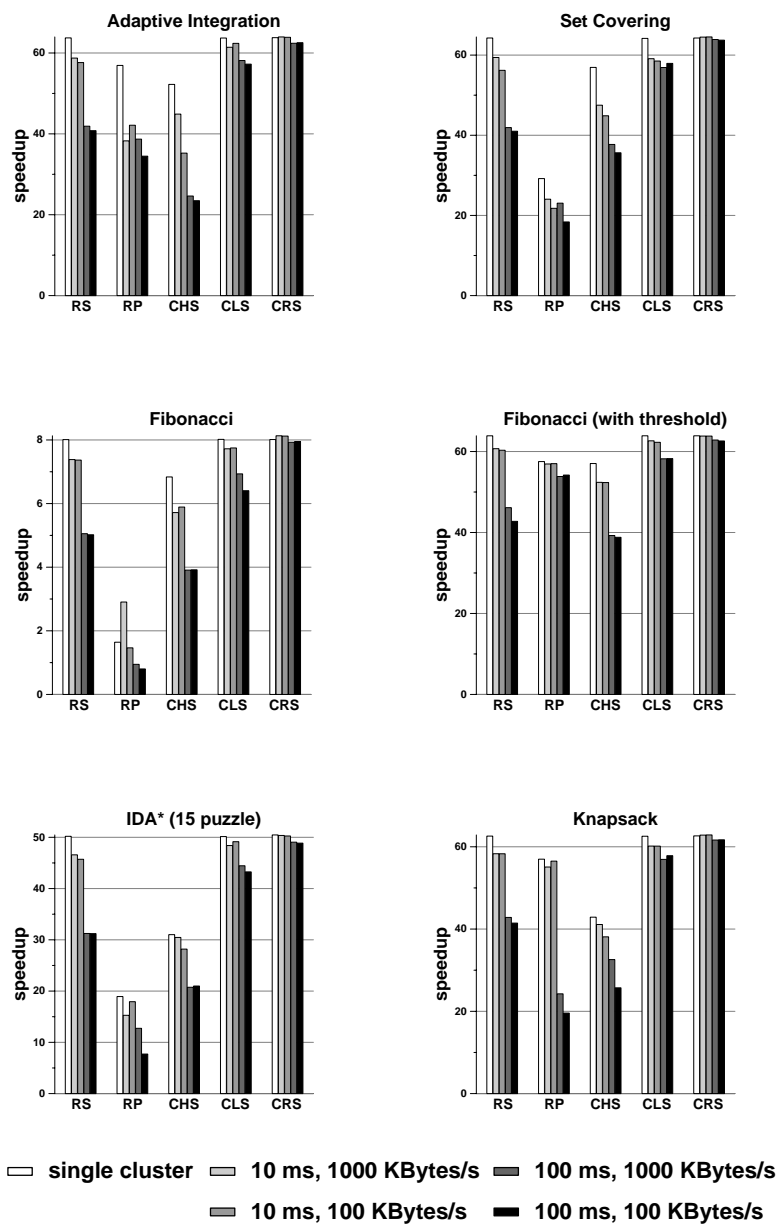
Figure 6.5: Speedups of 6 applications on 64 CPUs with 5 load-balancing algorithms and different wide-area latencies (one-way) and bandwidths (part 1).
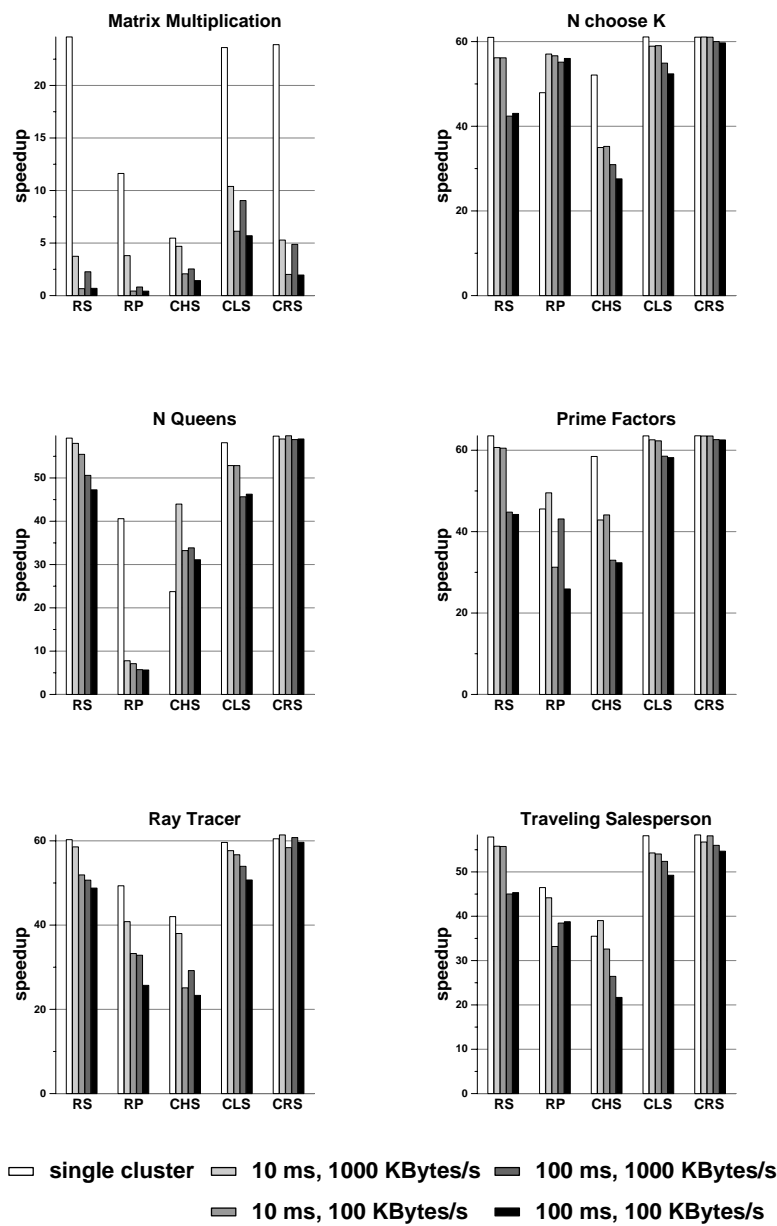
Figure 6.6: Speedups of 6 applications on 64 CPUs with 5 load-balancing algorithms and different wide-area latencies (one-way) and bandwidths (part 2).
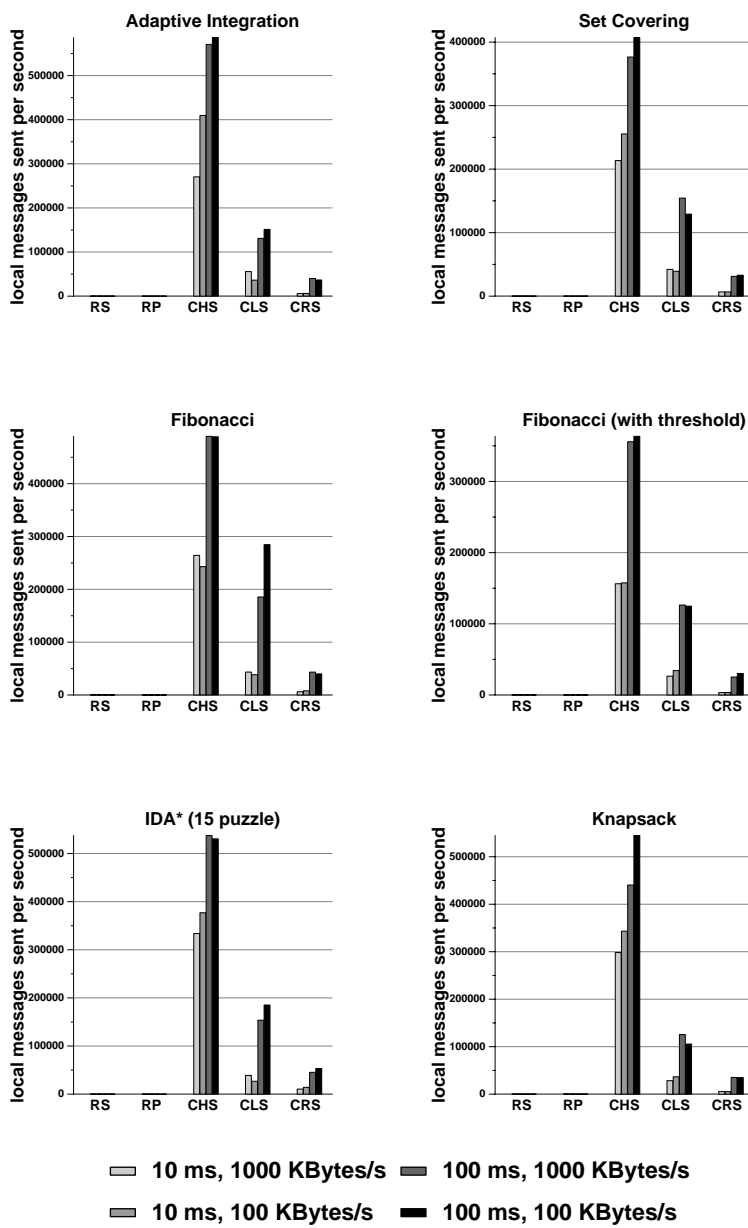
Figure 6.7: Total number of intra-cluster messages sent per second for 12 applications on 64 CPUs with 5 load-balancing algorithms and different wide-area latencies (one-way) and bandwidths (part 1).
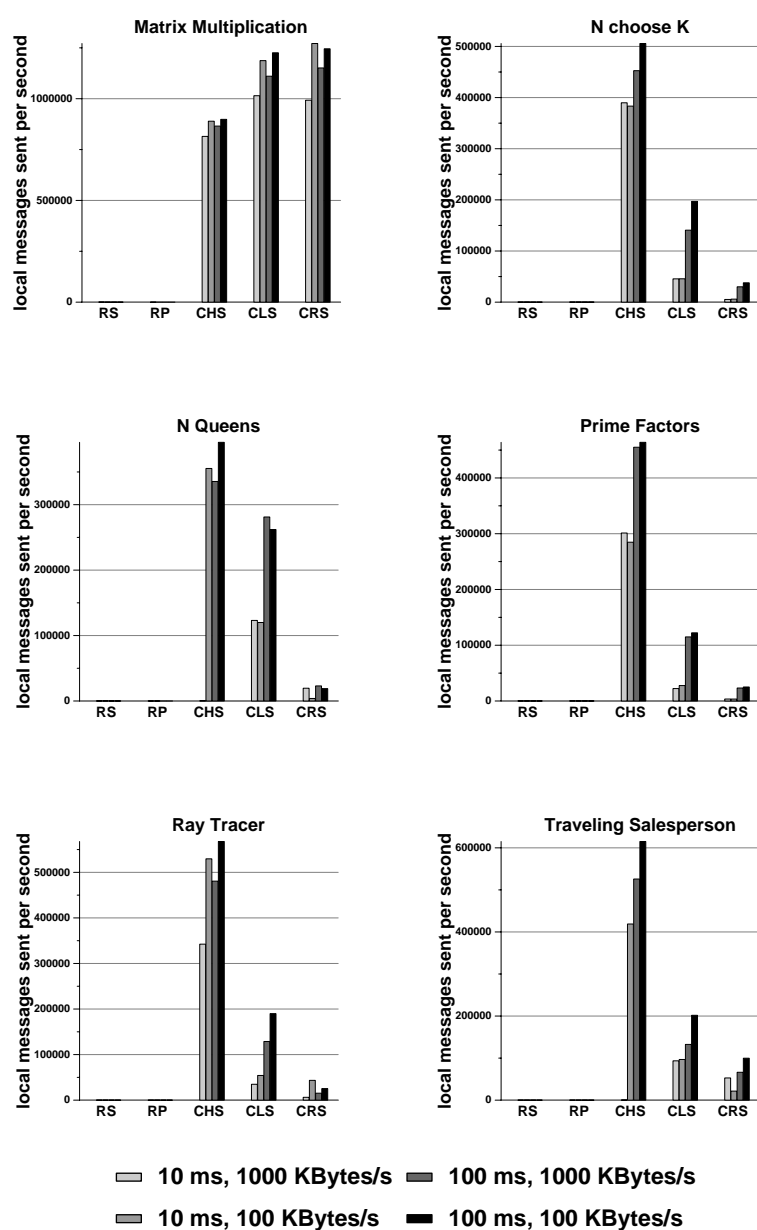
Figure 6.8: Total number of intra-cluster messages sent per second for 12 applications on 64 CPUs with 5 load-balancing algorithms and different wide-area latencies (one-way) and bandwidths (part 2).
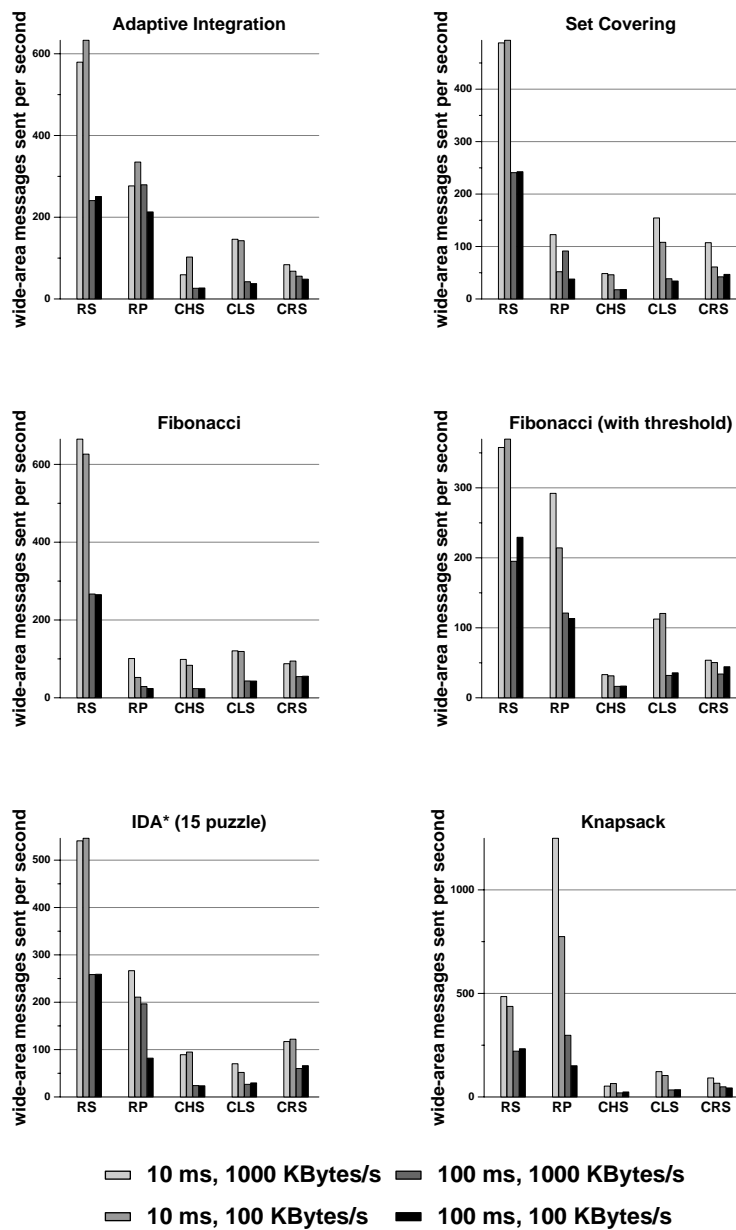
Figure 6.9: Total number of inter-cluster messages sent per second for 12 applications on 64 CPUs with 5 load-balancing algorithms and different wide-area latencies (one-way) and bandwidths (part 1).
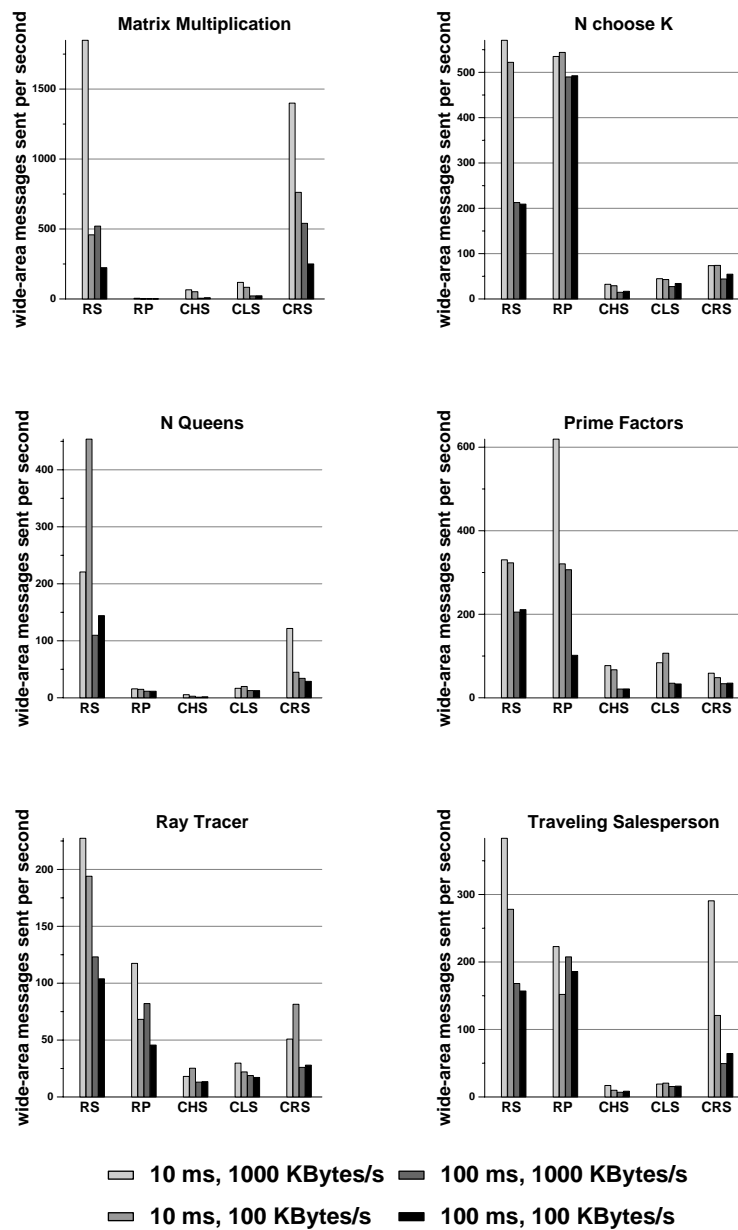
Figure 6.10: Total number of inter-cluster messages sent per second for 12 applications on 64 CPUs with 5 load-balancing algorithms and different wide-area latencies (one-way) and bandwidths (part 2).

6.2.2.1   Random Stealing (RS)

The performance of random stealing (RS) decreases considerably when wide-area laten-
cies and bandwidths are introduced, because, on four clusters, on average 76% of all steal
messages are sent to remote clusters (see Section 6.1.1). This can be seen in Figures 6.9
and 6.10: in most cases, RS sends more WAN messages than all other algorithms. Fig-
ures 6.5 and 6.6 show that nine out of twelve applications are almost solely bounded by
the network latency. Matrix multiplication, nqueens and raytracer are also bandwidth
sensitive.

   We collected statistics, which show that the speedups of the nine latency-bound ap-
plications can be completely explained with the number of wide-area messages sent. We
illustrate this with an example: with a one-way WAN latency of 100 milliseconds and a
bandwidth of 100 KBytes/s, adaptive integration sends in total 13051 synchronous steal
request messages over the wide-area links. Wide-area messages are also sent to return
results of stolen jobs to the victim nodes. Return messages are asynchronous, so they
do not stall the sender. In total, 1993 inter-cluster steal requests were successful, and
thus 1993 return messages are required. The total number of wide-area messages is
$13051 \times 2 + 1993 = 28095$ (steal attempts count twice, because there is both a request
and a reply). The run time of the application is 112.3 seconds, thus, as is shown in Fig-
ure 6.9, $28095/112.3 = 250.2$ wide-area messages per second are sent. The run time can
be explained using the wide-area steal messages: in total, the system has to wait 2610.2
seconds for steal reply messages (13051 times a round-trip latency of 0.2 seconds). Per
node, this is on average 40.8 seconds (2610.2 / 64). If we subtract the waiting time from
the run time we get $112.3 - 40.8 = 71.5$ seconds. The run time of adaptive integration on
a single cluster is 71.8 seconds, so the difference between the two run times is exactly the
waiting time that is caused by the wide-area steal requests.

6.2.2.2   Random Pushing (RP)

Random pushing (RP) shows varying performance. The results shown in Figures 6.5
and 6.6 use threshold values for the queue length that were manually optimized for
each application separately. For two applications (Fibonacci with spawn threshold and
n choose k), RP outperforms random stealing with high wide-area latencies, which is
due to its asynchronous communication behavior. For the other applications, however,
RP performs worse than RS. Moreover, our results indicate that a single threshold value
(specifying which jobs are executed locally, and which ones are pushed away) is not op-
timal for all cluster configurations. Finding the optimal values for different applications
and cluster configurations is a tedious task.

   With random pushing, nodes with empty queues are idle, and wait until they receive
a pushed job. We found that, for all applications, the idle times are quite large, and also
have a large deviation from the mean idle time (e.g., some nodes are hardly idle, others
are idle for many seconds). The idle times completely account for the high run times we
measured. We conclude that the bad performance of random pushing is caused by severe
load imbalance.

6.2.2.3   Cluster-aware Hierarchical Stealing (CHS)

The speedup graphs in Figures 6.5 and 6.6 show that cluster-aware hierarchical stealing already performs suboptimal in the single-cluster case. The additional statistics we gathered show that the cause is that CHS sends many more cluster-local messages than RS (see Figures 6.7 and 6.8). This is due to the high locality in the algorithm. This seems counterintuitive, but our statistics show that CHS transfers much more (small) jobs than the other algorithms. This is inherent to the algorithm, because all work in a subtree is always completed before messages are sent to the parent of the subtree. This causes many transfers of small jobs within the subtrees. For example, with the set covering problem on one cluster of 64 nodes, CHS transfers 40443 jobs, while RS only transfers 7734 jobs (about 19% of CHS).

Another indication of this problem is that nqueens and the traveling salesperson problem are both slower on one large cluster than on 4 clusters with low latencies. On a single cluster, CHS organizes the nodes in one large tree, while four smaller trees are used in the multi-cluster case, decreasing the amount of locality in the algorithm. In short, the locality of the algorithm causes very fine grained load balancing. RS does not suffer from this problem, because it has no concept of locality.

With multiple clusters, CHS sends fewer WAN messages per second than all other algorithms, as can be seen in Figures 6.9 and 6.10. However, this comes at the cost of idle time, which is visible in the speedup graphs (see Figures 6.5 and 6.6): for all applications, RS outperforms CHS. The problem is that CHS waits until the entire cluster is idle before wide-area steal attempts are done. This minimizes WAN traffic, but it also means that all nodes in the cluster must wait for the wide-area round-trip time, as only the cluster root initiates wide-area steal attempts. With RS, any node can do a wide-area steal attempt, and only the thief has to wait for the round-trip time.

Our results are confirmed by those achieved by Javelin 3 [120], which uses a hierarchical load-balancing algorithm, and also targets wide-area systems. In [120], speedup measurements are presented for TSP on a Linux cluster with 850 MHz Pentium IIIs. Even though the problem sizes used are extremely large (running between 12 and 21 hours on 64 machines), and the computation is extremely coarse grained (jobs ran on average for 177–430 seconds), speedups of only 24–40 were achieved on 64 machines (without any wide-area links).

The TSP version and problem size we used for our measurements runs only about 96 seconds with RS on a single cluster of 64 machines (816–462 times shorter), and has a much finer grain size: the Satin jobs ran only 8.166 milliseconds on average (i.e., they are about 5 orders of magnitude smaller). With CHS, Satin achieves similar speedups as reported by Javelin 3 (about 35.5 on 64 machines), but with RS, Satin performs much better, and achieves a speedup of 57.9 on 64 machines.

In a wide area setting (which is also Javelin 3's target platform), our measurements show that CHS is the worst performing algorithm for TSP, with a speedup of 21.7 on 64 machines (with a one-way WAN latency of 100 milliseconds and a throughput of 100 KBytes/s). The novel CRS algorithm performs much better, and still achieves a speedup of 56.7 on 64 machines (i.e., the application is about 2.5 times faster with CRS than with CHS). Unfortunately, no wide-area measurements are presented in [120].

6.2.2.4 Cluster-aware Load-based Stealing (CLS)

As can be seen in Figures 6.9 and 6.10, CLS sends more messages over the wide-area network than CHS, but performs much better (see Figures 6.5 and 6.6): CLS asynchronously prefetches work from remote clusters when the cluster load drops below the threshold. Also, it does not suffer from the locality problem as does CHS, because random stealing is used within a cluster. We ran all applications with load threshold values ranging from 1 to 64, but found that no single threshold value works best for all combinations of application, bandwidth and latency. The figures shown in this section give the results with the best cluster-load threshold for each application.

With high round-trip times, CLS sends fewer wide-area messages than CRS (see Figures 6.9 and 6.10), but it performs worse. However, with both algorithms, all inter-cluster communication is related to prefetching, and the load balancing within the clusters is the same. Also, both algorithms use the same location policy: they choose a random victim in a random remote cluster to prefetch work from. We conclude that the CLS coordinator cannot prefetch sufficient work for its worker CPUs; due to the high wide-area round-trip times, the stealing frequency is too low to acquire enough jobs.

It is also possible to adapt CLS to let all nodes multicast load information to all nodes in the local cluster. This would mean that all nodes can prefetch work, but the multicasting of load information is likely to have more impact on the performance than the unicast to the coordinator we use now. If the multicasts are done less frequently, prefetch decisions would be made with outdated load information. Moreover, multicasting the load information leads to scalability problems when large clusters are used.

6.2.2.5 Cluster-aware Random Stealing (CRS)

Our novel load-balancing algorithm, cluster-aware random stealing (CRS), performs best with all bandwidth/latency combinations, for all applications except matrix multiplication. Figures 6.5 and 6.6 show that CRS is almost completely insensitive to bandwidth and latency, although it sends more wide-area messages than CHS, and also more than CLS with high WAN latencies. CRS achieves superior performance because it neither has the locality problem of CHS, nor the prefetch limitation of CLS. With CRS, all nodes prefetch work over wide-area links, albeit only one job at a time per node. Statistics show that CRS prefetches up to 2.8 times the number of jobs prefetched by CLS. Moreover, CRS does not require any parameter tuning (such as the load threshold in CLS).

As shown in Figures 6.7 and 6.8, CRS does send three to four orders of magnitude more local messages than RS does. However, CRS sends the least number of cluster-local messages of all cluster-aware algorithms for all applications. Still, the measurements clearly indicate that CRS stresses the local network more than RS, and that efficient local communication thus is even more important on a wide-area system than on a single cluster. Also, it is clear that CRS exploits our assumption that wide-area systems are hierarchical. The results show that for these eleven applications, with four clusters, CRS has only at most 4% run-time overhead compared to a single, large Myrinet cluster with the same number of CPUs, even with a wide-area bandwidth of 100 KBytes/s and one-way WAN latency of 100 milliseconds.
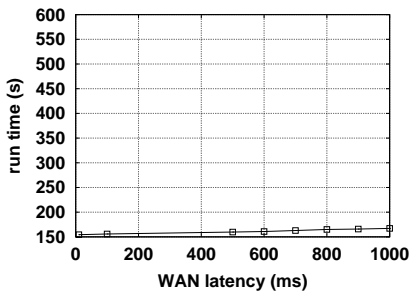
Figure 6.11: The effect of latency (one-way) on the raytracer, bandwidth is 100 KByte/s.
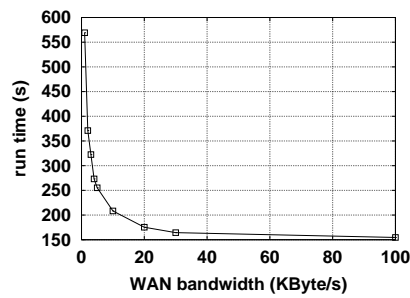
Figure 6.12: The effect of bandwidth on the raytracer, one-way latency is 100 ms.

6.3   Bandwidth-Latency Analysis

In this section we investigate the behavior of Satin's CRS algorithm under different bandwidth and latency conditions. We focus on the raytracer application, as it needs the most WAN bandwidth of all applications, except for matrix multiplication (which is not very interesting, as it is not suitable for wide-area systems).

Figure 6.11 shows the effect of latency on the raytracer application. It is clear that the CRS algorithm is insensitive to the WAN latency, as it is asynchronous. Even with a one-way WAN latency of one second, the run time of the raytracer with the CRS algorithm is only 167.1 seconds. For comparison, on one cluster of 64 nodes (i.e., without WAN links), the application takes 147.8 seconds. This means that even with WAN links with a one-way latency of one second, the overhead compared to a single cluster is only 13%.

The effect of WAN bandwidth on CRS is more significant, as is shown in Figure 6.12. The run times increase almost linearly, down to about 20 KByte/s. At this point, the speedup relative to the sequential version still is 50.8. When the bandwidth is further reduced, the run time increases exponentially. With a WAN bandwidth of only 1 KByte/s, the raytracer runs for 569.2 seconds, the speedup relative to the sequential version then is 15.6. Thus, with this low bandwidth, running on multiple clusters does not make sense anymore, as the same run time can be achieved on a single cluster of 16 machines. The other applications show similar behavior, but the bandwidth where the run times start to increase exponentially is even lower.

The results are promising, as CRS can tolerate very high WAN latencies. CRS also works well with low WAN bandwidths. However, there is a certain amount of WAN bandwidth that an application requires. If the bandwidth drops below that point, the run times increase exponentially. This bandwidth is very low for the applications we tried, 20 KByte/s for the application that requires the most WAN bandwidth (except for matrix multiplication). Furthermore, we believe that WAN bandwidth will become cheaper and cheaper, so this will become less of a problem. Latency however, is ultimately bound by the speed of light. It is therefore more important that CRS can tolerate high WAN latencies.
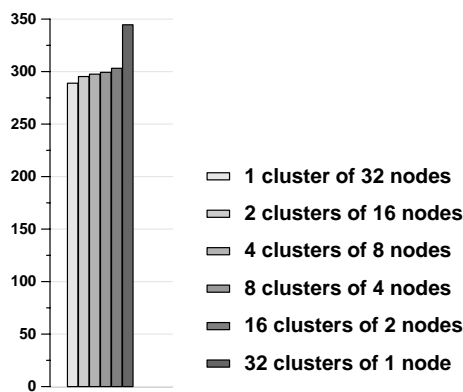
Figure 6.13: Run times of the raytracer on 32 nodes, with differently sized clusters. One-way latency used is 100 ms, bandwidth is 100 KByte/s.

6.4 Scalability Analysis of CRS

In this section, we investigate the scalability of CRS to more than four clusters. It is interesting to know how large the clusters should minimally be to achieve good performance with CRS on hierarchical wide-area systems. Therefore, we measured the performance of the raytracer (because this application uses the most WAN bandwidth) with differently sized clusters. We keep the total number of machines in the runs the same, so we can compare the run times.

We present measurements on 32 nodes, because of a limitation of the Panda cluster simulator. A gateway node is needed for each cluster. When many small clusters are used, many nodes are used as gateway, and can therefore not be used in the computation. As we only have 68 nodes in our cluster, we ran measurements with 32 nodes, because this way, we can run with 32 clusters of one node (thus, all links are WAN links), which is the most extreme scenario. With clusters of size one, the system is no longer hierarchical, so we can quantatively investigate how much the load balancing of the applications benefit from our assumption that systems are hierarchical in nature.

The run times of the raytracer with differently sized clusters are shown in Figure 6.13. The numbers show that CRS scales quite well to many small clusters. Even when using 16 clusters of only two nodes, the run time is only increased with 5% relatively to one single cluster (from 288.9 seconds to 303.1 seconds). When the system is no longer hierarchical (i.e., 32 clusters of one node), performance decreases considerably. In that case, CRS (like RS) uses only synchronous stealing, and is always idle during the WAN round-trip time.

We also investigated the performance of the raytracer on systems with differently sized clusters. Because of the way CRS selects steal targets in remote clusters (see the pseudo code for *remote_victim()* in Figure 6.2), all nodes in the system have the same chance of

being stolen from. This way, machines in smaller clusters are not favored over machines in larger clusters, and a good load balance is achieved. Compared to RS, CRS performs much better. When a system with two clusters, one of size 16, and one of size 48 is used (for a total of 64 nodes), with a one-way WAN latency of 100 ms and a bandwidth of 100 KByte/s, the speedup of RS is only 42.6, while CRS achieves a speedup of 54.3. The speedup is not as good as the speedup of CRS on four clusters with the same WAN link speed (59.7), because the total WAN bandwidth in the system is much lower (one link with 100 KByte/s, instead of six links of with 100 KByte/s). If the WAN bandwidth of the asymmetric system is increased to 600 KByte/s, the speedup of CRS increases to 58.4, only slightly less than the symmetric system, while RS achieves a speedup of 53.6.

When the system is made even more asymmetric, for instance a system with three clusters of 8 nodes and one cluster of 40 nodes (i.e., 64 in total), again with 100 ms one-way WAN latency and 100 KByte/s bandwidth, CRS still performs good with a speedup of 54.7, while RS achieves only 45.8.

6.5   Satin on the Grid: a Case Study

We now present a case study in which Satin runs across various emulated WAN scenarios. We use a testbed that emulates a grid on a single large cluster and supports various user-defined performance scenarios for the wide-area links of the emulated grid. We give a detailed performance evaluation of several load-balancing algorithms in Satin using this testbed.

We evaluate Satin's work-stealing algorithms by running four different applications across four emulated clusters. We use the following nine different WAN scenarios of increasing complexity, demonstrating the flexibility of Panda's WAN emulator. Figure 6.14 illustrates Scenarios 1–8 in detail.

1. The WAN is fully connected. The latency of all links is 100 ms (one-way); but the bandwidth differs between the links.

2. The WAN is fully connected. The bandwidth of all links is 100 KB/s; but the latency differs between the links.

3. The WAN is fully connected. Both latency and bandwidth differ between the links.

4. Like Scenario 3, but the link between clusters 1 and 4 drops every third second from 100 KB/s and 100 ms to 1 KB/s and 300 ms, emulating being busy due to unrelated, bursty network traffic.

5. Like Scenario 3, but every second all links change bandwidth and latency to random values between 10% and 100% of their nominal bandwidth, and between 1 and 10 times their nominal latency.

6. All links have 100 ms one-way latency and 100 KB/s bandwidth. Unlike the previous scenarios, two WAN links are missing, causing congestion among the different clusters.

7. Like Scenario 3, but two WAN links are missing.

8. Like Scenario 5, but two WAN links are missing.

9. Bandwidth and latency are taken from pre-recorded NWS measurements of the real DAS system.

In Scenarios 6–8, the traffic is routed around the missing links as follows. The data from cluster 1 to 3 is routed via cluster 2, while the data from cluster 4 to 2 is routed via cluster 1. Figure 6.15 shows the speedups achieved by four applications (adaptive integration, N queens, raytracer and TSP) on four clusters of 16 nodes each, with the WAN links between them being emulated according to the nine scenarios described above. For comparison, we also show the speedups for a single, large cluster of 64 nodes. Three work stealing algorithms, RS, CHS and CRS (described in Section 6.1) are compared with each other. We use RS and CHS, as they are used most often in the literature, and CRS, because it is our proposed solution. We omit CLS, as it always performs worse than CRS, and RP because the algorithm is unstable and uses too much memory, causing problems in some scenarios.

The most important scenario is Scenario 9, because that is a replay of NWS data, and thus is an indication of the performance of CRS in a real production system. We use a replay of measured data instead of running on the real wide-area systems, the results are deterministic this way, thus allowing a fair comparison between the load-balancing algorithms.

RS sends by far the most messages across the WAN links. The speedups it achieves are significantly worse, compared to a single, large cluster. This is especially the case in scenarios in which high WAN latency causes long idle times or in which low bandwidth causes network congestion. CHS is always the worst-performing algorithm, even within a single cluster, due to complete clusters being idle during a work-stealing message round-trip time.

CRS always is the best performing algorithm. Due to its limited and asynchronous wide-area communication, it can tolerate even very irregular WAN scenarios, resulting in speedups close to a single, large cluster. However, there are a few exceptions to the very high speedups achieved by CRS which occur whenever the WAN bandwidth becomes too low for the application's requirements. This happens with Scenarios 4, 8, and 9. But even in those cases, CRS still is Satin's best performing work-stealing algorithm.

The measurements show that CRS performs better on scenarios with missing links (i.e., Scenario 6 and 7) than scenarios with slow links (i.e., Scenario 4). This is caused by the routing of messages in the underlying system. When the simulator is configured with a non-fully connected network, the messages are routed around the missing links. When a link is just slow, this does not happen, and the slow link between two clusters is always used, instead of routing messages around the slow link.

A careful analysis of the scenarios where CRS performs suboptimally shows that CRS does not work well on systems that are highly asymmetric. We will explain this using two additional *extreme* Scenarios 10 and 11, which are shown in Figure 6.16. In these scenarios, there are two orders of magnitude between the slowest and the fastest WAN links, and the slow links have a bandwidth of only 1 KByte/s. The scenarios are not realistic, but
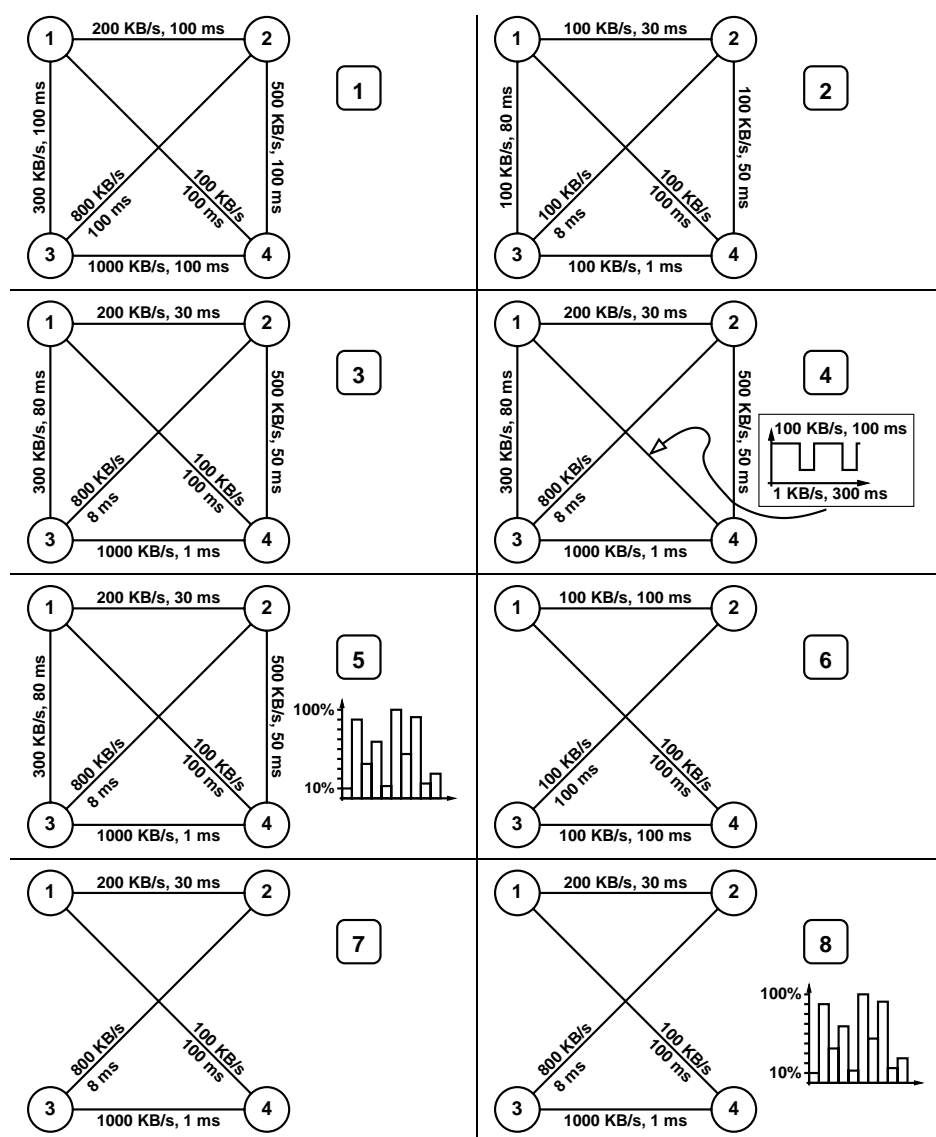
Figure 6.14: Emulated WAN Scenarios 1–8.
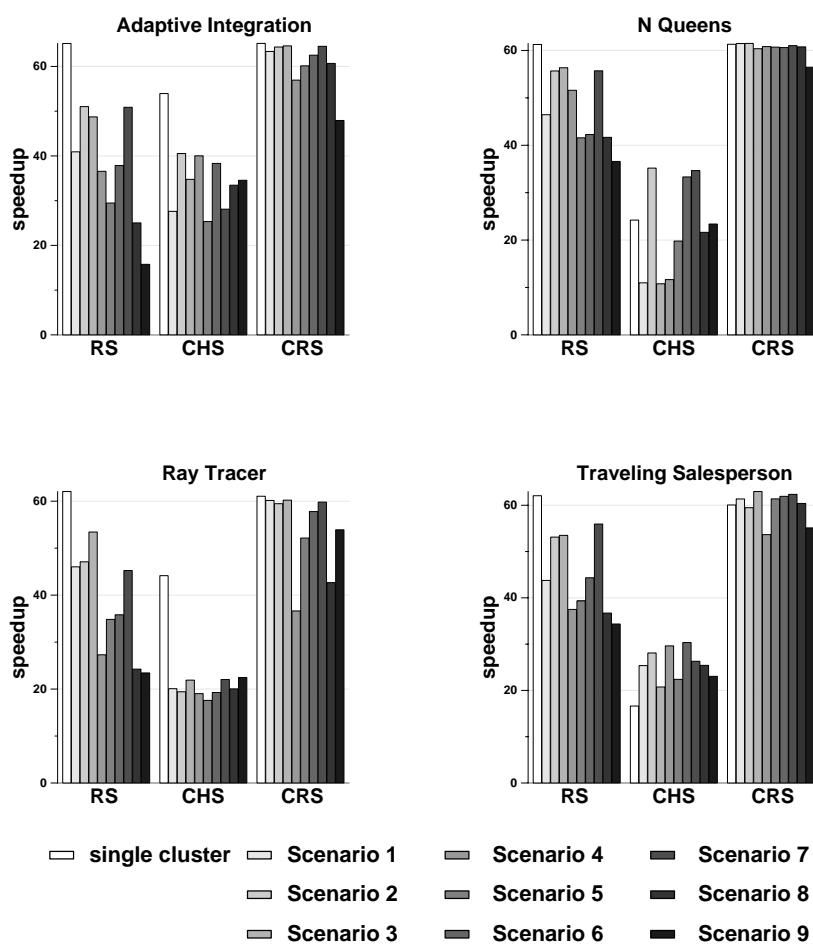
Figure 6.15: Speedups of 4 Satin applications with 3 load-balancing algorithms and 9 different, emulated WAN scenarios.
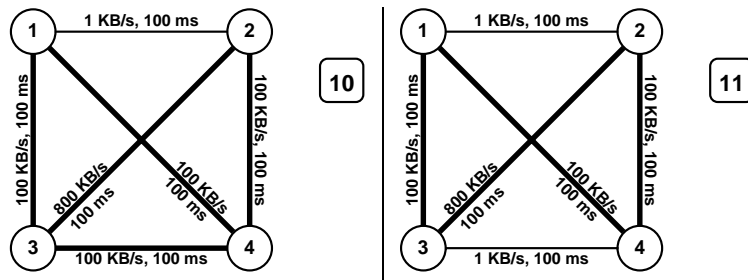
Figure 6.16: Emulated WAN Scenarios 10 and 11.

were deliberately chosen to emphasize the shortcomings of CRS in asymmetric systems. Scenario 10 is highly asymetic, because there is only one extremely slow link, between Clusters 1 and 2. Scenario 11 is similar to Scenario 10, but it is made symetrical by also using a slow link between Clusters 3 and 4. Thus, the total bandwidth of Scenario 11 is lower than that of Scenario 10. Therefore, one would expect that applications perform better on Scenario 10.

We investigate these scenarios using the raytracer, because this application uses the most WAN bandwidth (except for matrix multiplication). As shown in Section 6.3, the raytracer needs about 20 KByte/s of WAN bandwidth per link, therefore the slow WAN links are likely to become a bottleneck. The speedup of CRS with Scenario 10 is 19.9, while the speedup with Scenario 11 is 26.4. CRS achieves a lower speedup on Scenario 10, even though this scenario has more total bandwidth than Scenario 11.

This behavior can be explained as follows: when a node in Cluster 1 issues a wide-area steal attempt, it has a 33% chance of stealing from Cluster 2, as CRS uses random stealing. However, due to the low bandwidth between Cluster 1 and 2, the wide-area steals from a node in Cluster 1 to a node in Cluster 2 will take longer than a steal attempt from a node in Cluster 1 to a node in Cluster 3 or 4. Thus, nodes in Cluster 1 will spend a larger amount of time stealing from nodes in Cluster 2 than they will spend stealing from nodes in Clusters 3 and 4, even though the WAN links to Clusters 3 and 4 are faster. This leads to the underutilization of the fast WAN link between Clusters 1 and 3 and between 1 and 4, while the slow link between Cluster 1 and 2 is overloaded. To summarize: even though the chance that a node in Cluster 1 issues a wide-area steal to a node in Cluster 2 is 33%, the amount of time spent stealing from Cluster 2 is larger than 33% of the total wide-area stealing time, because steal attempts to nodes in Cluster 2 take longer than steal attempts to nodes in the other clusters.

This behavior results in load imbalance, as the nodes in Cluster 1 spend a large fraction of the time stealing over the slow WAN link, while nodes in Cluster 3 and 4 both spend 33% of their time stealing from nodes in Cluster 1 (both Clusters 3 and 4 have three WAN links, and all their links have the same speed). Thus, the nodes in Clusters 3 and 4 drain the work out of Cluster 1.

The same thing happens with Cluster 2: the work is drained by the nodes in the Clusters 3 and 4 while the nodes in Cluster 2 are stealing from nodes in Cluster 1, using

```
1  void cluster_aware_random_stealing(void) {
2    while(!exiting) {
3      job = queue_get_from_head();
4      if(job) {
5        execute(job);
6      } else {
7        if(nr_clusters > 1 && wan_credits_used < MAX_WAN_CREDITS) {
8          /* we have wan credits left, start wide-area steal */
9          wan_credits_used++;
10         send_async_steal_request(remote_victim());
11       }
12       /* do a synchronous steal in my cluster */
13       job = send_steal_request(local_victim());
14       if(job) queue_add_to_tail(job);
15     }
16   }
17 }
18
19 void handle_wide_area_reply(Job job, int victim) {
20   if(job) queue_add_to_tail(job);
21   wan_credits_used--;
22 }
```

Figure 6.17: Pseudo code for Cluster-aware Multiple Random Stealing.

the slow WAN link. The fact that Cluster 1 and 2 are lightly loaded leads to a downward spiral: it takes longer to steal over the slow WAN links, and the chance of getting work back is small, as the target cluster is being drained via the fast WAN links.

For highly dynamic or asymmetric grids, more sophisticated algorithms could achieve even better performance than CRS does. Some possible improvements on CRS are investigated in the remainder of this section.

6.5.1 Cluster-aware Multiple Random Stealing (CMRS)

To improve the performance of CRS on the aforementioned Scenarios 4, 8 and 9, we experimented with letting nodes send multiple wide-area steal requests to different clusters in parallel. The rationale behind this is that more prefetching of work, and prefetching from different locations, might alleviate the problem described above.

We implemented this as an extension of CRS. Each node gets a number of *WAN credits* to spend on wide-area steal requests. The pseudo code for CMRS is shown in Figure 6.17. CRS is a special case of CMRS: it is CMRS with only one WAN credit. CMRS works slightly better when the available credits are used for different clusters. This way, the algorithm does not spend multiple WAN credits to attempt to steal work from the same cluster. This optimization is not shown in Figure 6.17 for simplicity.

Measurements of the applications with CMRS show that it always performs worse than CRS. We believe this is the result of inefficient use of the available WAN bandwidth. Precious bandwidth is wasted on parallel wide-area steal request, while the first request may already result in enough work to keep the cluster busy. Moreover, CMRS does not

Cluster	t (ms)	1 / t	% chance of stealing from cluster
2	110	0.009	42.86
3	200	0.005	23.81
4	130	0.007	33.33
total	440	0.021	100.00

Table 6.2: An example of modifying the wide-area steal chances in ACRS.

solve the problem of asymmetry. In fact, it only worsens the situation, because the nodes in Cluster 3 and 4 also steal in parallel over the fast links. More prefetching drains the amount of work in Cluster 1 and 2 even faster.

6.5.2 Adaptive Cluster-aware Random Stealing (ACRS)

Another approach we investigated to improve the performance of CRS on the aforementioned Scenarios 4, 8 and 9, is actively detecting and circumventing slow links. Key is that the algorithm should balance *the amount of time* that is spent on steal requests to all remote clusters. The algorithm should also adapt to changes in WAN link speeds over time. We call the resulting algorithm Adaptive Cluster-aware Random Stealing (ACRS). Adaptivity can be implemented by changing the stochastic properties of the algorithm.

ACRS is almost identical to CRS. However, ACRS measures the time each wide-area steal request takes. The chances of sending a steal request to a remote cluster depends on the performance of the WAN link to that cluster (in this case, we use the time that the last steal request to that cluster took). Because we want the algorithm to prefer fast links over slow links, we do not calculate the chance distribution using the steal time, but use the inverse ($1/t$) instead.

$$chance_p = \frac{\frac{1}{t_p}}{\sum_i \frac{1}{t_i}} * 100\%$$

As shown in the formula above, the distribution is calculated by dividing the inverse time by the sum of all inverse times, and multiplying the result with 100%. An example is shown in Table 6.2. In this example, the distribution of probabilities of Cluster 1 in a system with four clusters is shown. The second column shows the time the last steal request to the destination cluster took. The numbers show that ACRS indeed has a preference for the fast links. Other performance data than the time the last steal request took can also be used, such as WAN-link performance data from the NWS.

The speedups of the four applications with ACRS on the nine different emulated WAN scenarios are shown in Figure 6.18. The slow WAN links are still a bottleneck (the chance of stealing over them is not zero), but the speedups of the raytracer with ACRS are significantly better than the speedups with CRS. Adaptive integration, N-queens and the traveling salesperson problem perform nearly optimal in all scenarios with ACRS. The speedup of adaptive integration with Scenario 9 (the WAN links are configured using NWS data), for instance, improves from 47.9 to 63.5. The raytracer, which is more bandwidth sensitive than the others, also improves with ACRS. The scenarios that performed
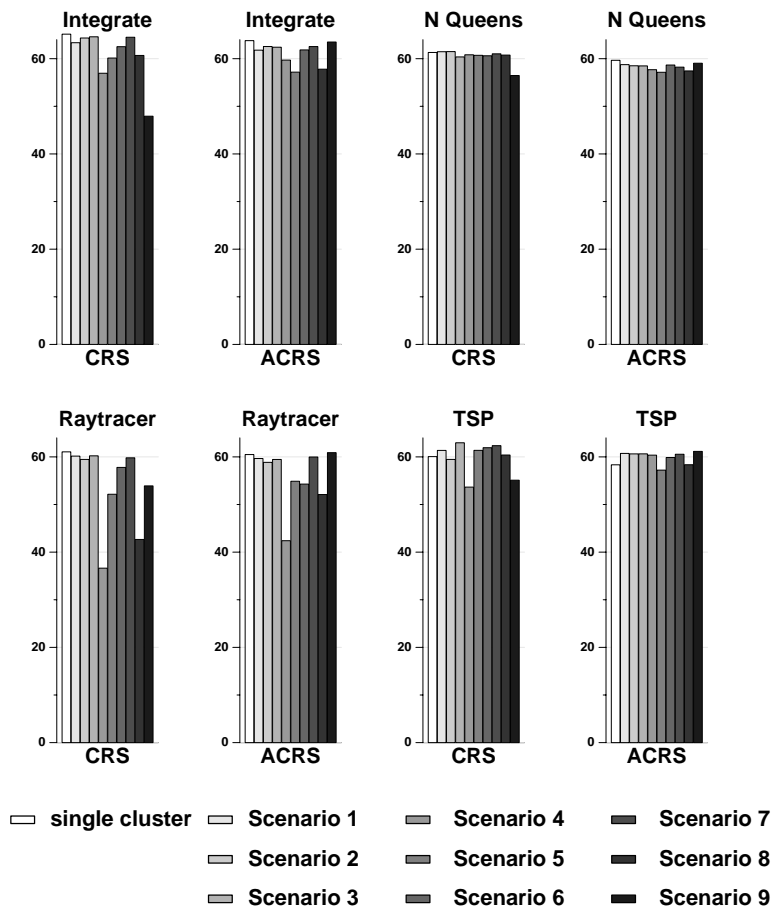
Figure 6.18: Speedups of 4 Satin applications with ACRS and 9 different, emulated WAN scenarios.

worst with CRS, 4, 8 and 9, perform significantly better with ACRS: the speedup of Scenario 4 improves from 36.6 to 42.4, Scenario 8 improves from 42.7 to 52.1, and Scenario 9 goes from a speedup of 53.9 to 60.1. The performance of the other scenarios is also slightly better. The performance of ACRS with the real-life Scenario 9, the playback of NWS data, is nearly perfect for all four applications.

ACRS also significantly improves Satin's performance on the two aforementioned extremely asymmetrical Scenarios 10 and 11 (See Figure 6.16). The speedup of the raytracer on Scenario 10 improves from 19.9 with the CRS algorithm to 30.2 with ACRS, while the speedup with Scenario 11 is increased from 26.4 to 29.5. These results show

Scenario	description	speedup CRS	speedup ACRS
9	NWS	53.9	60.1
10	asymmetric	19.9	30.2
11	symmetric	26.4	29.5

Table 6.3: Speedups for the raytracer with CRS and ACRS, with different scenarios.

that the ACRS performance does not suffer from the highly asymmetric systems as CRS does, because the speedup on Scenario 10 now is better than the speedup on Scenario 11, as can be expected, because the total bandwidth in Scenario 10 is larger.

There is a solution that might even perform better than ACRS. It is possible to use source routing for the WAN links at the Satin runtime system level. This way, the runtime system can route WAN steal messages around the slow links. With Scenario 10, for instance, messages from cluster one to cluster two could be routed via Cluster 3. However, this scheme also has several disadvantages. Besides WAN link performance data, global topology information is also needed for the routing of the messages. This information should be gathered during the run, because Internet routing tables may be updated at any time. This leads to a large run time overhead. Moreover, the system is much more complicated than CRS and ACRS, which perform sufficiently, except for very extreme, artificial scenarios.

6.6   Related Work

Another Java-based divide-and-conquer system is Atlas [14]. Atlas is a set of Java classes that can be used to write divide-and-conquer programs. Javelin 3 [120] provides a set of Java classes that allow programmers to express branch-and-bound computations, such as the traveling salesperson problem. Like Satin, Atlas and Javelin 3 are designed for wide-area systems. While Satin is targeted at efficiency, Atlas and Javelin are designed with heterogeneity and fault tolerance in mind, and both use a tree-based hierarchical scheduling algorithm. We found that this is inefficient for fine-grained applications and that *Cluster-aware Random Stealing* performs better. The performance of Javelin 3 was discussed in Section 6.1.3. Unfortunately, performance numbers for Atlas are not available at all.

Eager et al. [47] show that, for lightly loaded systems, sender-initiated load sharing (work pushing) performs better than receiver-initiated load sharing (work stealing). However, they assume zero latency and infinite bandwidth, which is inappropriate for wide-area systems. Furthermore, they describe a form of work pushing, where the sender first polls the target nodes, until a node is found that is not overloaded. Only after this node is found, the work is transferred. In a wide-area setting, this would imply that multiple wide-area round-trip times may be needed before work may be transferred. We used a different form of work pushing for the RP algorithm, where the work is pushed to a randomly-selected remote node, without further negotiations. If the receiving node is overloaded as well, it will immediately push the work elsewhere. The advantage of this scheme is that it is completely asynchronous.

Backschat et al. [8] describe a form of hierarchical scheduling for a system called Dynasty, targeting at large workstation networks. The algorithm is based on principles from economic theory, and requires intensive parameter tuning. However, the best-performing wide-area load-balancing method in Satin is not hierarchical within clusters, and requires no parameter tuning. Hence, it is much simpler to implement and it achieves better performance.

The AppLeS (short for application-level scheduling) project is an interesting initiative that provides a framework for adaptively scheduling applications on the grid. AppLeS focuses on selecting the best set of resources for the application out of the resource pool of the grid. Satin addresses the more low-level problem of load balancing the parallel computation itself, given some set of grid resources. AppLes provides (amongst others) a template for master-worker applications. Satin provides load balancing for the more general class of divide-and-conquer algorithms (see section 2.5).

An interesting model is explored by Alt et al. [2]. In their Java-based system, skeletons are used to express parallel programs. A special *DH* skeleton is provided that allows the programmer to express divide-and-conquer parallelism. Although the programming system targets grid platforms, it is not clear how scalable the approach is: in [2], measurements are provided only for a local cluster of 8 machines.

Network simulators like NSE [9] or DaSSF [107] focus on packet delivery and network protocols, rather than the network behavior as it is observed by an application. LAPSE (Large Application Parallel Simulation Environment) [45] simulates parallel applications on configurations with more than the available number of CPUs; the network behavior simulates the Intel Paragon machines. The MicroGrid software [155] virtualizes the grid resources like memory, CPU, and networks. For the simulation, all relevant system calls are trapped and mediated through the MicroGrid scheduler and the NSE network simulator. This approach goes further than Panda's network emulation, but also impacts the sequential execution of the application binaries. Panda's wide-area emulator, however, allows to run unmodified binaries of a parallel application, connecting them via physical LANs and emulated WANs. This network emulation provides a unique environment for experimentation with parallel applications on grid platforms, which has led to the development of our grid programming environments.

6.7   Conclusions

We have described our experience with five load-balancing algorithms for parallel divide-and-conquer applications on hierarchical wide-area systems. Our experimentation platform is Satin, which builds on the Manta high-performance Java system. Using 12 applications, we have shown that the traditional load-balancing algorithm used with shared-memory systems and workstation networks, random stealing, achieves suboptimal results in a wide-area setting.

We have demonstrated that hierarchical stealing, proposed in the literature for load balancing in wide-area systems, performs even worse than random stealing for our applications, because its load balancing is too fine grained, and because it forces all nodes of a cluster to wait while work is transferred across the wide-area network. Two other alterna-

tives, random pushing and load-based stealing, only work well after careful, application-specific tuning of their respective threshold values.

We introduced a novel load-balancing algorithm, called *Cluster-aware Random Stealing*. With this algorithm, 11 out of 12 applications are only at most 4% slower on four small clusters, compared to a large single-cluster system, even with high WAN latencies and low WAN bandwidths. These strong results suggest that divide-and-conquer parallelism is a useful model for writing distributed supercomputing applications on hierarchical wide-area systems.

A major challenge in investigating the performance of grid applications is the actual WAN behavior. Typical wide-area links are shared among many applications, making run time measurements irreproducible and thus scientifically hardly valuable. To allow a realistic performance evaluation of grid programming systems and their applications, we have used the Panda WAN emulator, a testbed that emulates a grid on a single, large cluster. This testbed runs the applications in parallel but emulates wide-area links by adding artificial delays. The latency and bandwidth of the WAN links can be specified by the user in a highly flexible way. This network emulation provides a unique environment for experimentation with high-performance applications on grid platforms.

We have used the Panda WAN emulator to evaluate the performance of Satin under many different WAN scenarios. The emulator allowed us to compare several load-balancing algorithms used by Satin under conditions that are realistic for an actual grid, but that are hard to reproduce on such a grid. Our experiments showed that Satin's CRS algorithm can actually tolerate a large variety of WAN link performance characteristics, and schedule parallel divide-and-conquer applications such that they run almost as fast on multiple clusters as they do on a single, large cluster. When CRS is modified to adapt to the performance of the WAN links, the performance of Satin on real-life scenarios (replayed NWS data) improves even more.

Now that we have shown, both with RMI and with Satin, that the Java-centric approach is indeed feasible, we will try to solve some of the remaining problems that Manta and Satin still have. The most important reason for a Java-centric grid-computing approach is that this way, the portability features of Java can be exploited. However, because Manta is a native system, this portability is at least partly lost. Therefore, we will try to apply the lessons we learned with Manta in a pure Java context in the next two chapters.

Chapter 7

Ibis: a Java-Centric Grid Programming Environment

```
One change leaves the way open for the introduction of others.
Whosoever desires constant success must change his conduct
with the times.

- Niccolo Machiavelli
```

Computational grids can integrate geographically distributed resources into a seamless environment [57]. In one important grid scenario, performance-hungry applications use the computational power of dynamically available sites. Here, compute sites may join and leave ongoing computations. The sites may have heterogeneous architectures, both for the processors and for the network connections. Running high-performance applications on such dynamically changing platforms causes many intricate problems. The biggest challenge is to provide a programming environment and a runtime system that combine highly efficient execution and communication with the flexibility to run on dynamically changing sets of heterogeneous processors and networks.

Although efficient message passing libraries, such as MPI, are widely used on the grid, they were not designed for such environments. MPI only marginally supports *malleable* applications (applications that can cope with dynamically changing sets of processors). MPI implementations also have difficulties to efficiently utilize multiple, heterogeneous networks simultaneously; let alone switching between them at run time. Moreover, MPI's programming model is targeted at SPMD style programs. MPI does not support upcalls, threads or RPCs. For grid computing, more flexible, but still efficient communication models and implementations are needed.

In the previous chapters, we have used the Manta system to investigate the usefulness of Java for grid computing. The fact that Manta uses a native compiler and runtime system allowed us to experiment with different programming models, implement and test several optimizations and to do detailed performance analysis. However, since our solutions were integrated into a native Java system, the useful "write once, run everywhere" feature of

Java is lost, while this feature was the reason to use Java for grid computing in the first place. In this chapter, we try to reimplement Manta's efficient serialization and communication mechanisms, but this time in pure Java, exploiting Java's "run everywhere" support. Even though, while developing Manta, we have identified Java's performance bottlenecks, this is still non-trivial, because we have to regard the JVM as a "black box"; we cannot modify it, as we did with Manta. The use of pure Java solves Manta's (and Satin's) portability problems, and makes it possible to run in environments with heterogenous networks and processors. One problem remains, however: in a grid environment, malleability is required. Manta does not support adding and removing machines in a running computation, as the underlying communication substrate (Panda) has a closed-world assumption. Our new communication implementation has been designed to operate in a grid environment and supports malleability, even when the underlying communication libraries do not.

In this chapter, we present a Java-based grid programming environment, called *Ibis* [1], that allows highly efficient communication in combination with any JVM. Because Ibis is Java-based, it has the advantages that come with Java, such as portability, support for heterogeneity and security. Ibis has been designed to combine highly efficient communication with support for both heterogeneous networks and malleability. Ibis can be configured dynamically at run time, allowing to combine standard techniques that work "everywhere" (e.g., using TCP) with highly-optimized solutions that are tailored for special cases, like a local Myrinet interconnect. Ibis should be able to provide communication support for any grid application, from the broadcasting of video to massively parallel computations. The *Ibis Portability Layer* (IPL), that provides this flexibility, consists of a set of well-defined interfaces. The IPL can have different implementations, that can be selected and loaded into the application *at run time*. We will call a loaded implementation an *Ibis instantiation*.

As a test case for our strategy, we implemented an optimized RMI system on top of Ibis. We show that, even on a regular JVM without any use of native code, our RMI implementation outperforms previous RMI implementations. When special native implementations of Ibis are used, we can run RMI applications on fast user-space networks (e.g., Myrinet), and achieve performance that was previously only possible with special native compilers and communication systems (e.g., Manta and Panda).

In the next chapter, we will describe a Satin implementation in pure Java, using the new serialization and communication implementation provided by Ibis. Satin will then exploit the portability features of Ibis to run divide-and-conquer programs on heterogenous and malleable systems.

The contributions of this chapter are the following:

1. The IPL provides an extensible set of interfaces that allow the construction of dynamic grid applications in a portable way.

2. The IPL communication primitives have been designed to allow efficient implementations.

---

[1] A sacred Ibis is a strongly nomadic, black and white bird that lives in large colonies throughout Africa, Australia and Asia.

3. The actual Ibis implementation to be used can be chosen at run time, by the application.

4. Using generated serialization code, object streaming, and a zero-copy protocol, Ibis makes object-based communication efficient.

5. We demonstrate the efficiency of Ibis with a fast RMI mechanism, implemented in pure Java.

6. The Ibis approach of combining solutions that work "everywhere" with highly optimized solutions for special cases provides efficiency and portability at the same time.

In the remainder of this chapter, we present the design of the Ibis system (Section 7.1). Section 7.2 explains two different Ibis implementations. We present a case study of Ibis RMI in Section 7.3. Finally, we draw our conclusions in Section 7.4.

7.1   Ibis Design

For deployment on the grid, it is imperative that Ibis is an extremely flexible system. Ibis should be able to provide communication support for grid applications, which can have widely different communication requirements: from the broadcasting of video to massively parallel computations. It should provide a unified framework for reliable and unreliable communication, unicasting and multicasting of data, and should support the use of any underlying communication protocol (TCP, UDP, GM, etc.) Moreover, Ibis should support malleability (i.e., machines must be able to join and leave a running computation).

Ibis consists of a runtime system and a bytecode rewriter. The runtime system implements the IPL. The bytecode rewriter is used to generate bytecode for application classes to actually use the IPL, a role similar to RMI's *rmic*. Below, we will describe how Ibis was designed to support the aforementioned flexibility, while still achieving high performance.

7.1.1   Design Goals of the Ibis Architecture

A key problem in making Java suitable for grid programming is how to design a system that obtains high communication performance while still adhering to Java's "write once, run everywhere" model. Current Java implementations are heavily biased to either portability or performance, and fail in the other aspect. Our strategy to achieve both goals simultaneously is to develop reasonably efficient solutions using standard techniques that work "everywhere", supplemented with highly optimized but non-standard solutions for increased performance in special cases. We apply this strategy to both computation and communication. Ibis is designed to use any standard JVM, but if a native optimizing compiler (e.g., Manta, see Chapter 3) is available for a target machine, Ibis can use it instead. Likewise, Ibis can use standard communication protocols (e.g., TCP/IP or UDP, as provided by the JVM), but it can also plug in an optimized communication substrate (e.g., GM or MPI) for a high-speed interconnect, if available. Essentially, our aim is to reuse all good ideas from the Manta native Java system (see Chapter 3), but now implemented in

pure Java. This is non-trivial, because Java lacks pointers, information on object layout, low-level access to the thread package inside the JVM, interrupts, and, before Java 1.4, a *select* mechanism to monitor the status of a set of sockets. The challenges for Ibis are:

1. how to make the system flexible enough to run seamlessly on a variety of different communication hardware and protocols;

2. how to make the standard, 100% pure Java case efficient enough to be useful for grid computing;

3. study which additional optimizations can be done to improve performance further in special (high-performance) cases.

Thus, grid applications using Ibis can run on a variety of different machines, using optimized software (e.g., a native compiler, the GM Myrinet protocol, MPI, etc.) where possible, and using standard software (e.g., TCP) when necessary. Inter operability is achieved by using the well-known TCP protocol: when multiple Ibis implementations are used (e.g., an Ibis implementation on top of MPI, and one on top of GM), all machines can still be used in one single computation, using TCP to communicate between the different implementations. Some underlying communication systems may have a closed world assumption. In that case, Ibis also uses TCP to glue multiple "closed worlds" together. Below, we discuss the three aforementioned issues in more detail.

7.1.1.1   Flexibility

The key characteristic of Ibis is its extreme flexibility, which is required to support grid applications. A major design goal is the ability to seamlessly plug in different communication substrates without changing the user code. For this purpose, the Ibis design uses the IPL, which consists of a small number of well-defined interfaces. The IPL can have different implementations, that can be selected and loaded into the application *at run time*.

The layer on top of the IPL can negotiate with Ibis instantiations through the well-defined IPL interface, and select the modules it needs. This flexibility is implemented using Java's dynamic class-loading mechanism. Although this kind of flexibility is hard to achieve with traditional programming languages, it is relatively straightforward in Java.

Many message-passing libraries such as MPI and GM guarantee reliable message delivery and FIFO message ordering. When applications do not require these properties, a different message-passing library might be used to avoid the overhead that comes with reliability and message ordering.

The IPL supports both reliable and unreliable communication, ordered and unordered messages, using a single, simple interface. Using user-definable properties (key-value pairs) applications can create exactly the communication channels they need, without unnecessary overhead.

7.1.1.2   Optimizing the Common Case

To achieve adequate performance with standard Java techniques, the main obstacle is communication performance. The problem of computational performance has already

been addressed sufficiently by other researchers. We will therefore use existing JVMs (e.g., those from IBM or Sun) and focus on communication performance.

To obtain acceptable communication performance, Ibis implements several optimizations. Most importantly, the overhead of serialization and reflection is avoided by generating special methods (in bytecode) for each object type at compile time. These methods can be used to convert objects to bytes (and vice-versa), and to create new objects on the receiving side, without using expensive reflection mechanisms. This way, the overhead of serialization is reduced dramatically.

Furthermore, our communication implementations use an optimized wire protocol. The Sun RMI protocol, for example, resends type information for each RMI. Our implementation caches this type information per connection. Using this optimization, our protocol sends less data over the wire, and equally important, saves processing time for encoding and decoding the type information.

7.1.1.3   Optimizing Special Cases

In many cases, the target machine may have additional facilities that allow faster computation or communication, which are difficult to achieve with standard Java techniques. One example we investigated in Chapter 3 is using a native, optimizing compiler instead of a JVM. This compiler (Manta), or any other high performance Java implementation, can simply be used by Ibis. We therefore focus on optimizing communication performance. The most important special case for communication is the presence of a high-speed local interconnect. Usually, specialized user-level network software is required for such interconnects, instead of standard protocols (TCP, UDP) that use the OS kernel. Ibis therefore was designed to allow other protocols to be plugged in. So, lower-level communication may be based, for example, on a locally-optimized MPI library. We have developed low-level network software based on Panda, which can likewise be used by Ibis. Panda is a portable communication substrate, which has been implemented on a large variety of platforms and networks, such as Fast Ethernet (using of UDP) and Myrinet (using GM). Panda was explained in more detail in Section 1.3.

An important issue we study in this chapter is the use of *zero-copy* protocols for Java. Such protocols try to avoid the overhead of memory copies, as these have a relatively high overhead with fast gigabit networks, resulting in decreased throughputs. With the standard serialization method used by most Java communication systems (e.g., RMI), a zero-copy implementation is impossible to achieve, since data is always serialized into intermediate buffers. With specialized protocols using Panda or MPI, however, a zero-copy protocol is possible for messages that transfer array data structures. For objects, the number of copies can be reduced to one. Another major source of overhead is in the JNI (Java Native Interface) calls [95] required to convert the floating point data types into bytes (and back). We found that, in combination with a zero-copy implementation, this problem can be solved by serializing into multiple buffers, one for each primitive data type. Implementing zero-copy (or single-copy) communication in Java is a nontrivial task, but it is essential to make Java competitive with systems like MPI for which zero-copy implementations already exist. The zero-copy Ibis implementation is described in more detail in Section 7.2.2.
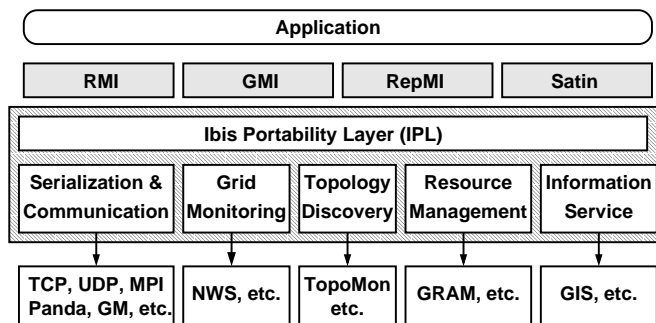
Figure 7.1: Design of Ibis. The various modules can be loaded dynamically, using run time class loading.

7.1.2 Design Overview

The overall structure of the Ibis system is shown in Figure 7.1. The grey box denotes the Ibis system. An important component is the IPL or *Ibis Portability Layer*, which consists of a set of Java interfaces that define how the layers above Ibis can make use of the lower Ibis components, such as communication and resource management. Because the components above the IPL can only access the lower modules via the IPL, it is possible to have multiple implementations of the lower modules. For example, we provide several communication modules. The layers on top of the IPL, however, make use of the communication primitives that the IPL defines, which are independent of the low-level communication library that is used. The IPL design will be explained in more detail in Section 7.1.3.

Below the IPL are the modules that implement the actual Ibis functionality, such as serialization, communication, and typical grid computing requirements, such as performance monitoring and topology discovery. Although serialization and communication are mostly implemented inside Ibis, this is not required for all functionality. For many components, standard grid software can be used. Ibis then only contains an interface to these software packages.

Generally, applications will not be built directly on top of the IPL (although this is possible). Instead, applications use some programming model for which an Ibis version exists. At this moment, we are implementing four runtime systems for programming models on top of the IPL: RMI, GMI, RepMI and Satin. RMI is Java's equivalent of RPC. However, RMI is object oriented and more flexible, as it supports polymorphism [167]. In Chapter 4 we demonstrated that parallel grid applications can be written with RMI. However, application-specific wide-area optimizations are needed. GMI [110] extends RMI with group communication. GMI also uses an object-oriented programming model, and cleanly integrates into Java, as opposed to Java MPI implementations. In the future, the wide-area optimizations for MPI (MagPIe [89]), will be ported to GMI, in order to make it more suitable for grid environments. RepMI extends Java with efficient replicated objects. Maassen et al. [111] show that RepMI also works efficiently on wide-area

systems. As explained in Chapters 5 and 6, Satin provides a programming model for divide-and-conquer and replicated-worker parallelism, and is specifically targeted at grid systems. The four mentioned programming models are integrated into one single system, and can be used simultaneously. A Satin program, for example, can make use of RMI and replicated objects (see also Section 5.2.5). In this chapter, we use our Ibis RMI implementation as a case study. The next chapter discusses the Satin implementation on top of Ibis, while we refer to [109] for the other programming models.

7.1.3   Design of the Ibis Portability Layer

The Ibis Portability Layer (IPL) is the interface between Ibis implementations for different architectures and the runtime systems that provide programming model support to the application layer. The IPL is a set of Java interfaces (i.e., an API). The philosophy behind the design of the IPL is the following: when efficient hardware primitives are available, make it possible to use them. Great care has to be taken to ensure that the use of mechanisms such as steaming data, zero-copy protocols and hardware multicast are not made impossible by the interface. We will now describe the concepts behind the IPL and we will explain the design decisions.

7.1.3.1   Negotiating with Ibis Instantiations

Ibis implementations are loaded at run time. A loaded Ibis implementation is called an Ibis instantiation. Ibis allows the application or runtime system on top of it to negotiate with the Ibis instantiations through the IPL, to select the instantiation that best matches the specified requirements. More than one Ibis implementation can be loaded simultaneously, for example to provide gateway capabilities between different networks. It is possible to instantiate both a Panda Ibis implementation that runs on top of a Myrinet network for efficient communication inside a cluster, and a TCP/IP implementation for (wide-area) communication between clusters.

Figure 7.2 shows example code that uses the IPL to load the best available Ibis implementation that meets the requirements of providing both FIFO ordering on messages and reliable communication. Although this code can be part of an application, it typically resides inside a runtime system for a programming model on top of the IPL. The *loadIbis* method is the starting point. First, it tries to load some user specified Ibis implementation, by calling *loadUserIbis*. If that fails, the code falls back to load the Panda implementation. If the Panda implementation cannot be loaded, the last resort is the TCP/IP implementation. The code in the *loadUserIbis* method shows how the layer above the IPL can negotiate with a loaded Ibis implementation about the desired properties. If the loaded Ibis implementation does not support the requirements, it can be unloaded. Subsequently, another implementation can be loaded and queried in the same way.

In contrast to many message passing systems, the IPL has no concept of hosts or threads, but uses location independent *Ibis identifiers* to identify Ibis instantiations. A registry is provided to locate communication endpoints, called *send ports* and *receive ports*, using Ibis identifiers. The communication interface is object oriented. Applications using the IPL can create communication channels between objects (i.e., ports), regardless

```
 1 Ibis loadUserIbis(String ibisImplName) {
 2     Ibis ibis = null;
 3     try {
 4         ibis = Ibis.createIbis("my ibis", ibisImplName);
 5     } catch (IbisException e) {
 6         return null; // Unable to load specified Ibis.
 7     }
 8
 9     // Implementation loaded. Does it provide the properties we need?
10     StaticProperties p = ibis.properties();
11     String ordering = p.find("message ordering");
12     if(!ordering.equals("FIFO")) {
13         ibis.unload();
14         return null; // FIFO message ordering is not supported.
15     }
16     String reliability = p.find("reliability");
17     if(!reliability.equals("true")) {
18         ibis.unload();
19         return null; // Reliable communication is not supported.
20     }
21     return ibis; // OK, return loaded implementation.
22 }
23
24 Ibis loadIbis(String ibisImplName) {
25     Ibis ibis = loadUserIbis(ibisImplName);
26     if(ibis == null) { // failed to load user Ibis
27         try {
28             ibis = Ibis.createIbis("my ibis",
29                 "ibis.ipl.impl.panda.PandaIbis");
30         } catch (IbisException e) { // Could not load Panda/Ibis.
31             try { // Fall back to TCP/Ibis.
32                 ibis = Ibis.createIbis("my ibis",
33                     "ibis.ipl.impl.tcp.TcpIbis");
34             } catch (IbisException e) {
35                 return null; // All failed!
36             }
37         }
38     }
39     return ibis; // Loaded implementation with the desired properties.
40 }
```

Figure 7.2: Loading and negotiating with Ibis implementations.

of the location of the objects. The connected objects can be located in the same thread, on different threads on the same machine, or they could be located at different ends of the world.

7.1.3.2   Send Ports and Receive Ports

The IPL provides communication primitives using send ports and receive ports. A careful design of these ports and primitives allows flexible communication channels, streaming

```
m = sendPort.getMessage();          send port    receive port       m = receivePort.receive();
m.writeInt(3);                                                        i = m.readInt();
m.writeIntArray(a);                                                  a = m.readIntArray();
m.writeIntSlice(b, 0, 100);                                          m.readIntSlice(b, 0, 100);
m.writeObject(o);                                                    o = m.readObject();
m.send();                                                            m.finish();
m.finish();
```
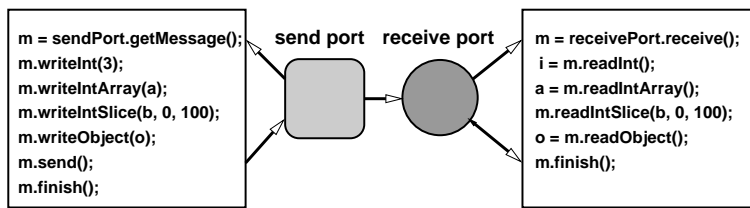
Figure 7.3: Send ports and receive ports.

of data, and zero-copy transfers. Therefore, the send and receive ports are important concepts in Ibis, and we will explain them in more detail below.

The layer above the IPL can create send and receive ports, which are then connected (the send port initiates the connection) to form a *unidirectional message channel*. Figure 7.3 shows such a channel. New (empty) message objects can be requested from send ports. Next, data items of any type can be inserted in this message. Both primitive types such as *long* and *double*, and objects, such as *String* or user-defined types, can be written. When all data is inserted, the *send* primitive can be invoked on the message, sending it off.

The IPL offers two ways to receive messages. First, messages can be received with the receive port's blocking *receive* primitive (see Figure 7.3). The *receive* method returns a message object, and the data can be extracted from the message using the provided set of read methods. Second, the receive ports can be configured to generate *upcalls*, thus providing the mechanism for implicit message receipt. The upcall provides the message that has been received as a parameter. The data can be read with the same read methods described above. The upcall mechanism is provided in the IPL, because it is hard to implement an upcall mechanism efficiently on top of an explicit receive mechanism [147]. Many applications and runtime systems rely on efficient implementations of upcalls (e.g., RMI, GMI and Satin). To avoid thread creation and switching, the IPL defines that there is at most *one* upcall thread running at a time per receive port.

Many message passing systems (e.g., MPI, Panda) are connection-less. Messages are sent to their destination, without explicitly creating a connection first. In contrast, the IPL provides a *connection-oriented* scheme (send and receive ports must be connected to communicate). This way, message data can be streamed. When large messages have to be transferred, the building of the message and the actual sending can be done simultaneously. This is especially important when sending large and complicated data structures, as can be done with Java serialization, because this incurs a large host overhead. It is thus imperative that communication and computation are overlapping.

A second design decision is to make the connections unidirectional. This is essential for the flexibility of the IPL, because sometimes it is desirable to have different properties for the individual channel directions. For example, when video data is streamed, the control channel from the client to the video server should be reliable. The return channel, however, from the video server back to the client, should be an unreliable channel with low jitter characteristics. For some applications there is no return channel at all (e.g.,
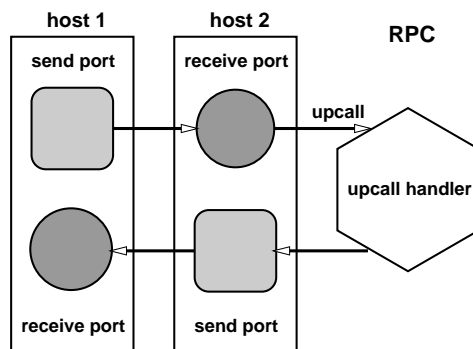
Figure 7.4: An RPC implemented with the IPL primitives.

wireless receivers that do not have a transmitter). The IPL can support all these communication requirements. Furthermore, on the Internet, the outbound and return channel may be routed differently. It is therefore sensible to make it possible to export separate outbound and return channels to the layers above the IPL when required. Recognizing that there are differences between the outbound and return channels is important for some adaptive applications or runtime systems, such as the Satin runtime system. Moreover, the IPL extends the send and receive port mechanism with multicast and many-to-one communication, for which unidirectional channels are more intuitive.

An important insight is that zero-copy can be made possible in some important special cases by carefully designing the interface for reading data from and writing data to messages. The standard Java streaming classes (used for serialization and for writing to TCP/IP sockets) convert all data structures to bytes, including the primitive types and arrays of primitive types. An exception is made only for byte arrays. Furthermore, there is no support for slices of arrays. When a pure Java implementation is used, the copying of the data is thus unavoidable. However, as can be seen in Figure 7.3, the IPL provides special read and write methods for (slices of) all primitive arrays. This way, Ibis allows efficient native implementations to support zero-copy for the array types, while only one copy is required for object types.

To avoid copying, the contract between the IPL and the layer above it is that the objects and arrays that are written to a message should not be accessed until a *finish* operation is called on the message. MPI and Panda define this in a similar way. For Figure 7.3, for instance, this means that it is the responsibility for the layer on top of the IPL to make sure that no other threads access the arrays a and b and the object o during the time between the write and the send operations.

Connecting send ports and receive ports, creating a unidirectional channel for messages is the *only* communication primitive provided by the IPL. Other communication patterns can be constructed on top of this model. By creating both a send and receive port on the same host, bidirectional communication can be achieved, for example to implement an RPC-like system, as Figure 7.4 shows. The IPL also allows a single send

multicast   **many to one**   **many to many**
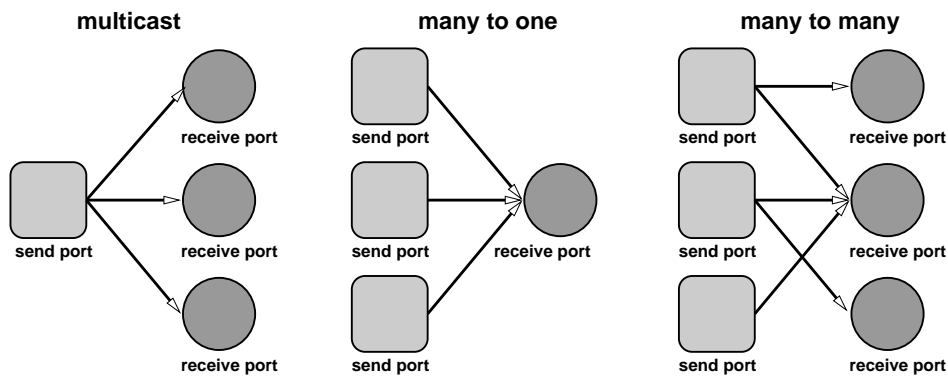
Figure 7.5: Other IPL communication patterns.

port to connect to multiple receive ports. Messages that are written to a send port that has multiple receivers are *multicast* (see Figure 7.5). Furthermore, multiple send ports can connect to a single receive port, thus implementing many-to-one and many-to-many communication (also shown in Figure 7.5).

The IPL defines that send ports can only be connected to a new receive port when no message is being constructed or sent. This way, race conditions are avoided, and the semantics are clear: the new receiver will only get all messages that are sent *after* the connection was made. Messages that were sent earlier are not delivered to the new receive port.

The IPL also defines clear semantics when a new send port tries to connect to a receive port that is already connected to one or more send ports. A receive port has only at most one outstanding message at any time: the receive port will deliver a new message only when the *finish* operation (see Figure 7.3) has been invoked on the previous message. When a new send port tries to connect to a receive port, the actual connection of the new sender is delayed until the receive port does not have an outstanding message (i.e., it is delayed until a *finish* operation). The IPL treats multiple connections to a single receive port as separate channels, and no message ordering is guaranteed between the channels, although the messages inside a single channel can be ordered.

For efficiency reasons, when additional senders or receivers connect to a port, the IPL does not guarantee anything about messages that were already sent, or about messages in transit. This means that it is the responsibility of the layer on top of the IPL to handle the synchronization between multiple senders and receivers on the same port.

7.1.3.3   Port Types

All send and receive ports that are created by the layers on top of the IPL are *typed*. Port types are defined and configured with properties (key-value pairs) via the IPL. Only ports of the same type can be connected. Port type properties that can be configured are, for example, the serialization method that is used, reliability, message ordering, performance

```
1  PortType myType;
2  StaticProperties p = new StaticProperties();
3  p.add("message ordering", "FIFO");
4  p.add("reliability", "true");
5  try {
6      myType = ibis.createPortType("my port type", p);
7  } catch (IbisException e) {
8      // Type creation fails when the Ibis instantiation
9      // does not provide the requested features.
10 }
11 try {
12     SendPort s = myType.createSendPort("my send port");
13 } catch (IbisException e) {
14     // Handle errors.
15 }
```

Figure 7.6: Creating and configuring a new port type.

monitoring support, etc. This way, the layers on top of the IPL can configure the send and receive ports they create (and thus the channels between them) in a flexible way. Figure 7.6 shows an example code fragment that creates and configures a port type, and creates a send port of this new type. Using the port type configuration mechanism, the software on top of the Ibis runtime system can negotiate with Ibis to obtain exactly the right functionality. This way, unnecessary overhead, (e.g., a reliability protocol when only unreliable communication is used) can be avoided.

The IPL does not exhaustively define which property names and values are legal. This way, the mechanism is flexible, and different implementations can use properties to get and set implementation-specific parameters (e.g., buffer sizes). It is evident that all implementations also have to agree upon some fixed set of common properties (e.g., properties describing basic functionality such as message ordering and reliability). The definition of this (small) common set is a part of the IPL.

7.1.4   Comparison with Other Communication Systems

Both TCP and UDP are widely used communication protocols that are also supported in Java. However, the programming interfaces (i.e., streams and unreliable datagrams) offered by the protocols are too low level for use in a parallel programming context. TCP does not provide multicast, which is important for many parallel applications. Although UDP does provide a multicast mechanism, it provides only unreliable communication, and no ordering is guaranteed between messages. However, parallel programs often require both properties.

Panda is an efficient communication substrate that was designed for runtime systems for parallel languages. However, as will be explained in Section 7.3, Panda's (and thus Manta RMI's) performance is hampered by the fact that Panda does not support the streaming of data. Moreover, Panda is not suitable for grid environments because it does not support malleability. Panda was already described in more detail in Section 1.3.

MPI [118] (Message Passing Interface) is a message passing library that is widely

feature / system	UDP	TCP	Panda	MPI	Nexus	IPL
FIFO ordering	no	yes	yes	yes	yes	yes
unordered messages	yes	no	no	yes	yes	yes
reliable messages	no	yes	yes	yes	yes	yes
unreliable messages	yes	no	no	no	yes	yes
streaming data	no	yes	no	no	no	yes
unordered multicast	yes	no	yes	yes	yes	yes
totally ordered multicast	no	no	yes	yes	yes	yes
malleability	yes	yes	no	no	yes	yes
multithreading support	no	no	yes	no	yes	yes
upcalls	no	no	yes	no	yes	yes
async upcalls (without polling)	no	no	yes	no	no	yes
explicit receive	yes	yes	no	yes	yes	yes
complex data structures	no	no	yes	yes	no	yes
multiple protocols	no	no	no	no	yes	yes
connection oriented	no	yes	no	no	yes	yes
multiple language bindings	yes	yes	no	yes	no	no

Table 7.1: Features of communication systems.

used for parallel programming. However, the programming model of MPI is designed for SPMD-style computations, and lacks support for upcalls and multithreading, which are essential for efficient RPC and RMI implementations. Moreover, MPI does not support the streaming of data, making transfers of complex data structures inefficient. MPI was initially targeted for shared-memory machines and clusters of workstations. Recently, however, there has been a large interest in using MPI in grid environments [52, 54, 55, 89]. MPI only marginally supports malleable applications. MPI implementations also have difficulties to efficiently utilize multiple, heterogeneous networks simultaneously, let alone switching between them at run time. For grid computing, more flexible, but still efficient communication models and implementations are needed.

Nexus [53] is a communication library in the Globus toolkit which, like Ibis, supports automatic selection of optimal protocols at run time. However, Nexus chooses the protocol based on the hardware it runs on, while Ibis is more flexible: it allows the layer on top of the IPL to negotiate about the protocol that should be used. This is important, because the optimal protocol can depend on the application, not only on the hardware that is available.

Nexus is written in C, and is therefore portable only in the traditional sense. The interface provided by Nexus is similar to the IPL: the only supported mechanism is a connection-oriented one-way channel that allows asynchronous messages. However, Nexus does not support asynchronous upcalls (i.e., polling is required to trigger upcalls), and more importantly, Nexus does not provide a streaming mechanism, as a single data buffer is passed to the asynchronous message when it is sent. We found that the streaming of data is imperative to achieve good performance with complex data structures. The performance results provided in [53] show that Nexus adds considerable overhead to the low-level communication mechanisms upon which is built.

The extreme flexibility of the IPL is shown by the features that are supported in the interface. Table 7.1 lists several features of communication systems, and compares the

```
1  class Foo implements java.io.Serializable {
2      int i1, i2;
3      double d;
4      int[] a;
5      Object o;
6      String s;
7      Foo f;
8  }
```

Figure 7.7: An example serializable class: *Foo*.

IPL with UDP, TCP, Panda, MPI and Nexus. The great strength of the IPL is that all features can be accessed via one single interface. The layers above the IPL can enable and disable features as wanted using key-value pairs (properties). Moreover, the IPL is extensible, as more properties can be added without changing the interface.

7.2 Ibis Implementations

In this section, we will describe the existing Ibis implementations. An important part is the implementation of an efficient serialization mechanism. Although the Ibis serialization implementation was designed with efficient communication in mind, it is independent of the lower network layers, and completely written in Java. The communication code, however, has knowledge about the serialization implementation, because it has to know how the serialized data is stored to avoid copying. Applications can select at run time which serialization implementation (standard Sun serialization or optimized Ibis serialization) should be used for each individual communication channel. At this moment we have implemented two communication modules, one using TCP/IP and one using message-passing (*MP*) primitives. The MP implementation can currently use Panda, while an MPI implementation is work in progress. The TCP/IP implementation is written in 100% pure Java, and runs on any JVM. The MP implementation requires some native code.

7.2.1 Efficient Serialization

Serialization is a mechanism for converting (graphs of) Java objects to some format that can be stored or transferred (e.g., a stream of bytes, or XML). Serialization can be used to ship objects between machines. One of the features of Java serialization is that the programmer simply lets the objects to be serialized implement the empty, special "marker" interface *java.io.Serializable*. Therefore, no special serialization code has to be written by the application programmer. For example, the *Foo* class in Figure 7.7 is tagged as serializable in this way. The serialization mechanism makes a deep copy of the objects that are serialized. For instance, when the first node of a linked list is serialized, the serialization mechanism traverses all references, and serializes the objects they point to (i.e., the whole list). The serialization mechanism can handle cycles, making it possible to convert arbitrary data structures to a stream of bytes. When objects are serialized, not only the object data is converted into bytes, but type and version information is also added. When the

stream is deserialized, the versions and types are examined, and objects of the correct type can be created. When a version or type is unknown, the deserializer can use the bytecode loader to load the correct class file for the type into the running application. Serialization performance is of critical importance for Ibis (and RMI), as it is used to transfer objects over the network (e.g., parameters to remote method invocations for RMI). Serialization was discussed in more detail in Section 2.3.2.

In Chapter 3, we described what the performance bottlenecks in the serialization mechanism are, and how it can be made efficient when a native Java system is used (e.g., Manta). The most important sources of overhead in standard Java serialization are run time type inspection, data copying and conversion, and object creation. Because Java lacks pointers, and information on object layout is not available, it is non-trivial to apply the techniques we implemented for Manta. We will now explain how we avoid these sources of overhead in the Ibis serialization implementation.

7.2.1.1   Avoiding Run Time Type Inspection

The standard Java serialization implementation uses run time type inspection, called *reflection* in Java, to locate and access object fields that have to be converted to bytes. The run time reflection overhead can be avoided by generating serialization code for each class that can be serialized. This way, the cost of locating the fields that have to be serialized is pushed to compile time. Ibis provides a bytecode rewriter that adds generated serialization code to class files, allowing all programs to be rewritten, even when the source code is not available. The rewritten code for the *Foo* class from Figure 7.7 is shown in Figure 7.8 (we show Java code instead of bytecode for readability).

The bytecode rewriter adds a method that writes the object fields to a stream, and a constructor that reads the object fields from the stream into a newly created object. A constructor must be used for the reading side, because all *final* fields must be initialized in all constructors. It is impossible to assign *final* fields in a normal method. Furthermore, the *Foo* class is tagged as rewritten with the same mechanism used by standard serialization: the bytecode rewriter lets *Foo* implement the empty *ibis.io.Serializable* marker interface.

The generated write method (called *ibisWrite*) just writes the fields to the stream one at a time. The constructor that reads the data is only slightly more complicated. It starts with adding the *this* reference to the cycle check table, and continues reading the object fields from the stream that is provided as parameter. The actual handling of cycles is done inside the Ibis streaming classes. As can be seen in Figure 7.8, strings are treated in a special way. Instead of serializing the *String* object, the value of the string is directly written in the UTF-8[2] format, which is more compact than a character array, because Java uses Unicode strings with two bytes per character.

7.2.1.2   Optimizing Object Creation

The reading side has to rebuild the serialized object tree. In general, the exact type of objects that have to be created is unknown, due to inheritance. For example, the *o* field in the

---

[2]a transformation format of Unicode and ISO 10646

```
1  public final class FooGenerator extends Generator {
2      public final Object doNew(ibis.io.IbisInputStream in)
3        throws ibis.ipl.IbisIOException, ClassNotFoundException {
4          return new Foo(in);
5      }
6  }
7
8  class Foo implements java.io.Serializable,
9                        ibis.io.Serializable {
10     int i1, i2;
11     double d;
12     int[] a;
13     String s;
14     Object o;
15     Foo f;
16
17     public void ibisWrite(ibis.io.IbisOutputStream out)
18       throws ibis.ipl.IbisIOException {
19         out.writeInt(i1);
20         out.writeInt(i2);
21         out.writeDouble(d);
22         out.writeObject(a);
23         out.writeUTF(s);
24         out.writeObject(o);
25         out.writeObject(f);
26     }
27
28     public Foo(ibis.io.IbisInputStream in)
29       throws ibis.ipl.IbisIOException,
30       ClassNotFoundException {
31         in.addObjectToCycleCheck(this);
32         i1 = in.readInt();
33         i2 = in.readInt();
34         d = in.readDouble();
35         a = (int[])in.readObject();
36         s = in.readUTF();
37         o = (Object)in.readObject();
38         f = (Foo)in.readObject();
39     }
40 }
```

Figure 7.8: Rewritten code for the *Foo* class.

Foo object (see Figure 7.8) can refer to any non-primitive type. Therefore, type descriptors that describe the object's class have to be sent for each reference field. Using the type descriptor, an object of the actual type can be created. Standard Java serialization uses an undocumented native method inside the standard class libraries to create objects without invoking a constructor (it may have side effects). We found that this is considerably more expensive than a normal *new* operation.

   Ibis serialization implements an optimization that avoids the use of this native method, making object creation cheaper. For each serializable class, a special *generator class*

```
1  int typeDescriptor = readInt();
2
3  Generator gen = getFromGeneratorTable(typeDescriptor);
4  if (gen == null) { // Type encountered for the first time.
5      String name = readClassName();
6      gen = Class.newInstance(name + "Generator");
7      addToGeneratorTable(typeDescriptor, gen);
8  }
9
10 return gen.doNew();
```

Figure 7.9: Pseudo code for optimized object creation.

(called *FooGenerator* in the example in Figure 7.8) is generated by the bytecode rewriter. The generator class contains a method with a well-known name, *doNew*, that can be used to create a new object of the accompanying serializable class (*Foo*).

Pseudo code that implements the object creation optimization using the generator class is shown in Figure 7.9. When a type is encountered for the first time, the *IbisInput-Stream* uses the standard (but expensive) *Class.newInstance* operation to create an object of the accompanying *generator* class (lines 5–6). A reference to the generator object is then stored in a table in the *IbisInputStream* (line 7). When a previously encountered type descriptor is read again, the input stream can do a cheap lookup operation in the generator table to find the generator class for the type (line 3), and create a new object of the desired class, by calling the *doNew* method on the generator (line 10). Thus, the *IbisInputStream* uses *newInstance* only for the first time a type is encountered. All subsequent times, a cheap table lookup and a normal *new* is done instead of the call to a native method that is used by standard Java serialization. Side effects of user-defined constructors are avoided because *doNew* calls the *generated* constructor that reads the object fields from the input stream. A user-defined constructor is never invoked. This optimization effectively elimi-nates a large part of the object creation overhead that is present in standard serialization.

The serialization mechanism is further optimized for serializable classes that are *final* (i.e., they cannot be extended). When a reference field to be serialized points to a final class, the type is known at compile time, and no type information is written to the stream. Instead, the deserializer directly does a *new* operation for the actual class. Example code, generated by the bytecode rewriter, assuming that the *Foo* class is now final, is shown in Figure 7.10. The code only changes for the *f* field, because it has the (now final) type *Foo*. The generated code for the other fields is omitted for brevity. The *ibisWrite* method now directly calls the *ibisWrite* method on the *f* field. The *writeKnownObjectHeader* method handles cycles and *null* references. On the reading side, a normal *new* operation is done, as the type of the object that has to be created is known at compile time.

The bytecode rewriter usually is run at compile time, before the application program is executed, but it can in principle also be applied at run time to classes that are loaded dynamically. This can happen for instance when classes are loaded over the network.

```
1  public final void ibisWrite(ibis.io.IbisOutputStream out)
2    throws ibis.ipl.IbisIOException {
3      // Code to write fields i1 ... o is unchanged.
4      boolean nonNull = out.writeKnownObjectHeader(f);
5      if(nonNull) f.ibisWrite(out); // Call generated serializer.
6  }
7
8  public Foo(ibis.io.IbisInputStream in)
9    throws ibis.ipl.IbisIOException, ClassNotFoundException {
10     // Code to read fields i1 ... o is unchanged.
11     int opcode = in.readKnownTypeHeader();
12     if(opcode == 0) f = new Foo(in); // Normal object.
13     else if(opcode > 0) f = (Foo)in.getFromCycleCheck(opcode);
14
15     // If none of the above cases it true, the reference is null.
16     // No need to assign field: Java initializes references to null.
17 }
```

Figure 7.10: Optimization for *final* classes.

7.2.1.3 Avoiding Data Copying

Ibis serialization tries to avoid the overhead of memory copies, as these have a relatively
high overhead with fast networks, resulting in decreased throughputs (see Section 3.5.4).
With the standard serialization method used by most RMI implementations, a zero-copy
implementation is impossible to achieve, since data is always serialized into intermediate
buffers. By using special typed buffers and treating arrays separately, Ibis serialization
achieves zero-copy for arrays, and reduces the number of copies for complex data struc-
tures to one. The generated serialization code uses the *IbisInputStream* and *IbisOutput-
Stream* classes to read and write the data. We will now explain these classes in more detail
using Figure 7.11. The streaming classes use *typed buffers* to store the data in the stream
for each primitive type separately. When objects are serialized, they are decomposed into
primitive types, which are then written to a buffer of the same primitive type. No data
is converted to bytes, as is done by the standard object streams used for serialization in
Java. Arrays of primitive types are handled specially: a reference to the array is stored
in a separate array list, no copy is made. The stream data is stored in this special way to
allow a zero-copy implementation (which will be described in Section 7.2.2).

 Figure 7.11 shows how an object of class *Bar* (containing two integers, a double, and
an integer array) is serialized. The *Bar* object and the array it points to are shown in the
upper leftmost cloud labeled "application heap". As the *Bar* object is written to the output
stream, it is decomposed into primitive types, as shown in the lower cloud labeled "Ibis
runtime system heap". The two integer fields *i1* and *i2* are stored in the *integer* buffer,
while the double field *d* is stored in the separate *double* buffer. The array *a* is not copied
into the typed buffers. Instead, a *reference* to the array is stored in a special array list.
When one of the typed buffers is full, or when the *flush* operation of the output stream
is invoked, the data in *all* typed buffers is streamed. The output stream will write all
typed buffers to the stream, and not only the buffers that are full, because all data items
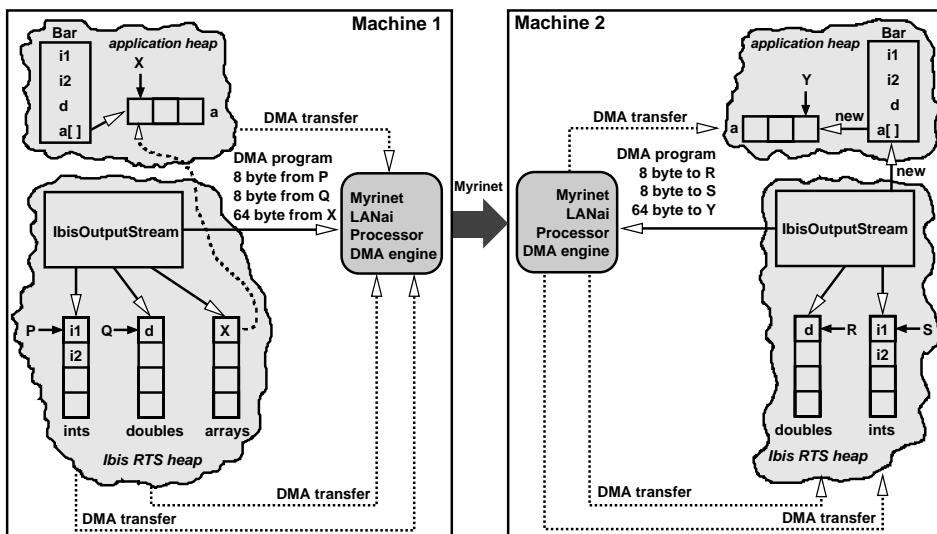
Figure 7.11: Low-level diagram of zero-copy data transfers with the Ibis implementation on Myrinet.

in the message are read in the same order they were packed. Hence, all data that was stored in any of the buffers is needed. Next, the array list will be traversed. How this is implemented depends on the lower I/O layer, and will be explained in more detail in the next section.

For implementations in pure Java, all data has to be converted into bytes, because that is the only way to write data to TCP/IP sockets, UDP datagrams and files[3]. Therefore, the typed buffer scheme is limited in its performance gain in a pure Java implementation. All types except floats and doubles can be converted to bytes using Java code. For floats and doubles, Java provides a native method inside the class libraries (*Float.floatToIntBits* and *Double.doubleToLongBits*), which must be used for conversion to integers and longs, which can subsequently be converted to bytes. Because these native methods must be provided by all JVMs, their use does not interfere with Ibis' portability requirements. When a float or double array is serialized, a native call is used for conversion *per element*. Because the Java Native Interface is quite expensive (see [95]), this is a major source of overhead for both standard and Ibis serialization. However, this overhead is unavoidable without the use of additional native code: using the typed buffers mechanism and some native code, the conversion step from primitive types to bytes can be optimized by converting all typed buffers and all primitive arrays in the array list using *one* native call. For native communication implementations, conversion may not be needed altogether; the typed buffer scheme then allows a zero-copy implementation. On the receiving side, the

---

[3]Since Java version 1.4, it is possible to avoid data conversion with the new I/O interface (the java.lang.nio package).

	Sun serialization		Ibis with conversion		Ibis zero-copy	
	read	write	read	write	read	write
data structure / JVM	Sun 1.4					
100 KB byte[]	99.6		115.9	∞	134.2	∞
100 KB int[]	64.3	151.0	77.6	282.3	118.4	∞
100 KB double[]	32.7	38.4	36.2	43.3	135.1	∞
1023 node tree, user payload	8.2	12.4	41.7	50.6	61.0	71.0
1023 node tree, total data	11.8	17.9	63.0	76.4	92.2	107.3
data structure / JVM	IBM 1.31					
100 KB byte[]	97.0		121.6	∞	120.4	∞
100 KB int[]	68.2	144.9	82.4	186.4	110.4	∞
100 KB double[]	54.8	61.0	49.9	45.2	121.2	∞
1023 node tree, user payload	3.3	6.0	31.7	49.0	38.6	80.5
1023 node tree, total data	4.7	8.6	47.9	74.0	58.3	121.7
data structure / JVM	Manta					
100 KB byte[]	171.3		187.8	∞	195.3	∞
100 KB int[]	87.2	113.6	68.8	195.3	160.1	∞
100 KB double[]	85.7	120.6	54.3	111.6	203.5	∞
1023 node tree, user payload	5.9	15.6	23.0	35.5	36.5	60.0
1023 node tree, total data	8.6	22.4	34.7	53.3	55.2	90.8

Table 7.2: Ibis serialization throughput (MByte/s) for three different JVMs.

typed buffers are recreated, as is shown in the right side of Figure 7.11. The read methods of the input stream return data from the typed buffers. When a primitive array is read, it is copied directly from the data stream into the destination array. The rest of Figure 7.11 (i.e., the communication part) will be explained in the Section 7.2.2.

7.2.1.4   Ibis Serialization Performance

We ran several benchmarks to investigate the performance that can be achieved with the Ibis serialization scheme. All measurements were done on the DAS-2 system (see Section 1.9.2). The results for Sun and Ibis serialization with different Java systems are shown in Table 7.2. The table gives the throughput of serialization to memory buffers, thus no communication is involved. The results give an indication of the host overhead caused by serialization, and give an upper limit on the communication performance. We show serialization and deserialization numbers separately, as they often take place on different machines, potentially in parallel (i.e., when data is streamed). Due to object creation and garbage collection, deserialization is the limiting factor for communication bandwidth. This is reflected in the table: the throughput numbers for serialization (labeled "write") are generally higher than the numbers for deserialization (labeled "read").

For Ibis serialization, two sets of numbers are shown for each JVM: the column labeled "conversion" includes the conversion to bytes, as is needed in a pure Java implementation. The column labeled "zero-copy" does not include this conversion and is representative for Ibis implementations that use native code to implement zero-copy communication. We present results for arrays of primitive types and balanced binary trees with nodes that contain four integers. For the trees, we show both the throughput for

the user payload (i.e., the four integers) and the total data stream, including type descriptors and information that is needed to rebuild the tree (i.e., the references). The latter gives an indication of the protocol overhead. Some numbers are infinite (the ∞-signs in the table), because zero-copy implementations just store references to arrays in the array list. This overhead is independent of the size of the arrays, making throughput numbers meaningless. The same holds for serialization of byte arrays, as these are not converted.

The numbers show that data conversion is expensive: throughputs without conversion (labeled zero-copy) are much higher. It is clear that, without the use of native code, high throughputs can be achieved with Ibis serialization, especially for complex data structures. For reading binary trees, for instance, Ibis serialization with data conversion achieves a payload throughput that is 5.0 times higher than the throughput of standard serialization on the Sun JIT, and 9.6 times higher on the IBM JIT. When data conversion is not needed, the difference is even larger: the payload throughput for Ibis serialization is 7.4 times higher on the Sun JIT and 11.6 times higher on the IBM JIT.

With Sun serialization on Manta, the throughput for *int* and *double* arrays is higher than with Ibis serialization. The reason is that Manta's native Sun serialization implementation avoids zeroing new objects on the reading side. This is possible because the runtime system knows that the array data will be overwritten with data from the stream. With Ibis serialization, the Manta runtime system cannot know this, and has to initialize them to their default value (i.e., false, zero, or null).

7.2.2   Efficient Communication

It is well known [147, 162] that in (user level) communication systems most overhead is in software (e.g., data copying). Therefore, much can be gained from software optimization. In this section, we will describe the optimizations we implemented in the TCP/IP and Panda Ibis implementations. The general strategy that is followed in both implementations is to avoid thread creation, thread switching, locking, and data copying as much as possible.

7.2.2.1   TCP/IP Implementation

The TCP/IP Ibis implementation is relatively straightforward. One socket is used per unidirectional channel between a single send and receive port. Because Ibis' communication primitives are connection oriented, the sockets remain connected as long as the send and receive ports are connected. It is not necessary to create a new socket connection for each individual message. However, we found that using a socket as a one-directional channel is inefficient. This is caused by the flow control mechanism of TCP/IP. Normally, acknowledgment messages are piggybacked on reply packets. When a socket is used in only one direction, there are no reply packets, and acknowledgments cannot be piggybacked. Only when a timeout has expired are the acknowledgments sent in separate messages. This severely limits the throughput that can be achieved. Because it is common that a runtime system (or application) creates both an outgoing and a return channel (for instance for RMI, see Figure 7.4), it is possible to optimize this scheme. Ibis implements channel pairing: whenever possible, the outgoing and return data channels are combined and use

only one socket. This optimization is transparent for the programmer (it is not reflected in the IPL) and greatly improves the throughput. An additional advantage is the reduced number of file descriptors that is used by the optimized scheme.

A socket is a one-to-one connection. Therefore, with multicast or many-to-one communication (e.g., multiple clients connecting to one server via RMI), multiple sockets must be created. A problem related to this is that Java initially did not provide a *select* primitive. It has been recently added in Java 1.4, in the *java.nio* package. The *select* operation can be used on a set of sockets, and blocks until data becomes available on any of the sockets in the set. We use Ibis for grid computing, and the latest features of Java may not be available on all platforms. A *select* operation cannot be implemented in native code, because Java does not export file descriptors. Therefore, we must also provide a solution when *select* is not present. In that case, there are only two ways to implement many-to-one communication (both are supported by Ibis).

First, it is possible to poll a single socket, using the method *InputStream.available*. A set of sockets can thus be polled by just invoking *available* multiple times, once per socket. However, the *available* method must do a system call to find out whether data is available for the socket. Hence, it is an expensive operation. Moreover, polling wastes CPU time. This is not a problem for single-threaded applications that issue explicit receive operations (e.g., MPI-style programs), but for multithreaded programs this is undesirable. When implicit receive is used in combination with polling, CPU time is always wasted, because one thread must be constantly polling the network to be able to generate upcalls.

Second, it is possible to use one thread per socket. Each thread calls a blocking receive primitive, and is unblocked by the kernel when data becomes available. This scheme does not waste CPU time, but now each thread uses memory space for its stack. Moreover, a thread switch is needed to deliver messages to the correct receiver thread when explicit receive is used. Ibis allows the programmer to decide which mechanism should be used via the properties mechanism described in Section 7.1.3.

An important limitation of using TCP/IP is that a file descriptor is associated with a socket. Many operating systems allow only a limited number (in the order of several hundreds) of file descriptors. Therefore, the use of TCP/IP limits the scalability of this particular Ibis implementation. We intend to avoid this problem in the future by providing an Ibis implementation on top of UDP instead of TCP. The UDP protocol does not suffer from this problem (because it is not connection-oriented). Also, it is supported in standard Java, therefore no native code is required. However, a UDP implementation is more complicated, as UDP does not offer reliable data transmission, fragmentation and flow control. Currently, the programming models that are implemented on top of the IPL require these features.

7.2.2.2   Zero-Copy Message-Passing Implementation

The message-passing (*MP*) implementation is built on top of native message-passing libraries (written in C), such as Panda and MPI. For each send operation, the typed buffers and application arrays to be sent are handed as a message fragment to the MP device, which sends the data out without copying; this is a feature supported by both Panda and MPI. This is shown in more detail in Figure 7.11. On the receive side, the typed fields

JVM	Sun 1.4		IBM 1.31		Manta	
network	TCP	GM	TCP	GM	TCP	GM
latency downcall	143	61.5	134	33.3	147	35.1
latency upcall	129	61.5	126	34.4	140	37.0
Sun serialization						
100 KB byte[]	9.7	32.6	10.0	43.9	9.9	81.0
100 KB int[]	9.4	28.2	9.6	42.5	9.1	27.6
100 KB double[]	8.4	18.7	9.1	29.5	9.0	27.6
1023 node user	2.8	3.2	1.7	2.7	1.4	1.9
1023 node total	4.0	4.6	2.4	3.9	2.0	2.7
Ibis serialization						
100 KB byte[]	10.0	60.2	10.3	123	10.0	122
100 KB int[]	9.9	60.2	9.6	122	9.7	122
100 KB double[]	9.0	60.2	9.2	123	9.7	123
1023 node user	5.9	17.7	5.8	23.6	4.4	22.0
1023 node total	8.9	26.7	8.8	35.6	6.6	33.2

Table 7.3: Ibis communication round-trip latencies in μs and throughputs in MByte/s on TCP (Fast Ethernet) and GM (Myrinet).

are received into pre-allocated buffers; no other copies need to be made. Only when the sender and receiver have different byte ordering, a single conversion pass is made over the buffers on the *receiving* side. Arrays are received in the same manner, with zero-copy whenever the application allows it.

As with the TCP/IP implementation, multiplexing of Ibis channels over one device is challenging to implement efficiently. It is difficult to wake up exactly the desired receiver thread when a message fragment arrives. For TCP/IP, there is support from the JVM and kernel that manage both sockets and threads: a thread that has done a receive call on a socket is woken up when a message arrives on that socket. A user-level MP implementation has a more difficult job, because the JVM gives no hooks to associate threads with communication channels. An Ibis implementation might use a straightforward, inefficient solution: a separate thread polls the MP device, and triggers a thread switch for each arrived fragment to the thread that posted the corresponding receive. We opted for an efficient implementation by applying heuristics to maximize the chance that the thread that pulls the fragment out of the MP device actually is the thread for which the fragment is intended. A key observation is that a thread that performs an explicit receive operation is probably expecting to shortly receive a message (e.g., a reply). Such threads are allowed to poll the MP device in a tight loop for roughly two latencies (or until their message fragment arrives). After this polling interval, a *yield* call is performed to relinquish the CPU. A thread that performs an upcall service receives no such privileged treatment. Immediately after an unsuccessful poll operation, it yields the CPU to another polling thread.

7.2.2.3 Performance Evaluation

Table 7.3 shows performance data for both Ibis implementations. For Myrinet, we use the Ibis MP implementation on top of Panda, which in turn uses GM for low-level commu-

platform	Ibis/IBM 1.31		MPI/C	
network	TCP	GM	TCP	GM
latency downcall	134	33.3	161	22
latency upcall	126	34.4	–	–
100 KB byte[]	10.3	123	11.1	143
1023 node user	5.8	23.6	5.4/5.8	16.6/30.0

Table 7.4: Low-level IPL and MPI communication latencies in μs and throughputs in MByte/s on TCP (Fast Ethernet) and GM (Myrinet).

nication. For comparison, we show the same set of benchmarks as in Table 7.2 for both Sun serialization and Ibis serialization. The latency numbers shown are round-trip times for an empty message. Because Ibis provides both explicit and implicit message receipt, both are shown separately in the table (labeled downcall and upcall respectively).

Ibis is able to exploit the fast communication hardware and provides low communication latencies and high throughputs on Myrinet, especially when arrays of primitive types are transferred. The numbers show that Ibis provides portable performance, as all three Java systems achieve high throughputs with Ibis serialization. On Fast Ethernet, Ibis serialization only gains up to 8% throughput for primitive arrays. The difference is small, because Sun serialization is already able to utilize almost the complete network bandwidth of 100 Mbit/s (12.5 MByte/s). On Myrinet, however, the performance gain of Ibis serialization is considerable. Because Ibis serialization implements zero-copy transfers for arrays of primitive types and can do in-place receipt, thus avoiding *new* operations, the throughput for arrays is a factor of 1.5–4.5 higher (depending on the type of the array and the JVM). For complex data structures the differences are even larger, because the generated serialization code and the typed-buffers mechanism used by Ibis are much more efficient than the run time type inspection used by Sun serialization. On Fast Ethernet, Ibis serialization is a factor of 2.2–3.4 faster (depending on the JVM), while on Myrinet, Ibis serialization wins by a even larger amount: Ibis is a factor of 5.5–11.6 faster.

The array throughput on Myrinet (GM) for the Sun JIT is low compared to the other Java systems, because the Sun JIT makes a copy when a pointer to the array is requested in native code, while the other systems just use a pointer to the original object. Because this copying affects the data cache, throughput for the Sun JIT is better for smaller messages of 40KB, where 83.4 MByte/s is achieved.

7.2.2.4   Performance Comparison

Table 7.4 provides results for both IPL-level communication and MPI. On Fast Ethernet, we used MPICH-p4, on Myrinet MPICH-GM. For MPI, only downcall latencies are shown (explicit receipt), as MPI does not support upcalls (implicit receipt). The performance of IPL-level communication is close to MPI, although the latency on Myrinet currently still is 50% higher than for MPI. This is partly due to the JNI overhead for invoking the native code (GM) from Java. Also, the Ibis communication protocol is currently optimized for throughput, not latency, and adds 20 bytes of overhead to each message. On Fast Ethernet, the Ibis latency actually is better than the MPI latency.

	Sun RMI	KaRMI 1.05b		Ibis RMI		Manta RMI	
network	TCP	TCP	GM	TCP	GM	TCP	GM
null-latency	218.3	127.9	32.2	131.3	42.2	127.5	34.8
array throughput							
100 KB byte[]	9.5	10.3	57.2	10.3	76.0	10.1	113.0
100 KB int[]	9.5	9.6	45.6	9.6	76.0	10.1	113.0
100 KB double[]	10.2	9.5	25.1	9.1	76.0	10.1	113.0
tree throughput							
1023 node user	2.2	2.3	2.5	4.3	22.9	3.7	6.9
1023 node total	3.2	3.3	3.6	6.5	34.6	5.6	10.5

Table 7.5: RMI latencies in μs and throughputs in MByte/s on TCP (Fast Ethernet) and GM (Myrinet) using the IBM JIT. For comparison, we also show Manta native RMI.

For the MPI binary tree throughput, two different measurements were done, one letting the receiver create the tree data structures dynamically (as in the Java benchmark), and one that reuses the same data structure for every message. The latter optimization can, in general, not be used in Java, since the exact type of the received data is not known in advance (due to polymorphism). Because Ibis has support for streaming, it can overlap serialization, sending and deserialization of complex data structures. This significantly increases the throughput on a slow network like Ethernet, where Ibis performs as well as the fastest MPI version. Even on the fast Myrinet network, the Ibis tree communication still has a higher throughput than its 'dynamic' MPI counterpart.

7.3   A case study: Efficient RMI

As described in Section 7.1.1, we are implementing four runtime systems as a part of Ibis. In this chapter, we focus on the RMI implementation, because RMI is present in standard Java, and many people are familiar with its programming model. Satin on top of Ibis will be discussed in the next chapter. The API of Ibis RMI is identical to Sun's RMI. RMI is straightforward to implement, because most building blocks are present in the IPL. We extended the bytecode rewriter (which we use to generate serialization code) to generate the RMI stubs and skeletons. The RMI registry is implemented on top of the IPL registry. Communication channels are set up as shown in Figure 7.4. Thus, each stub has a send port, and each skeleton creates a receive port. When an RMI program issues a *bind* operation, the ports are connected. Using the properties mechanism described in Section 7.1.3, the ports can be configured to use either Sun serialization or Ibis serialization.

Table 7.5 shows the latencies and throughputs that are achieved by several RMI implementations on our hardware using the IBM JIT. KaRMI [130] is an optimized serialization and RMI implementation. On TCP, both KaRMI and Ibis RMI are is implemented in pure Java. There also is a KaRMI version that uses native code to interface with GM, which makes KaRMI a good candidate for performance comparison with Ibis RMI, both on TCP and GM. For comparison, Table 7.5 also shows numbers for Manta RMI (see Chapter 3), using the native Manta compiler instead of the IBM JIT.

As can be seen, RMI has a substantial performance overhead on Myrinet, compared to IPL-level communication (see Table 7.3). The higher latency is mainly caused by thread creation and thread switching at the receiving side, which are a result of the RMI model: RMI uses implicit message receipt (pop-up threads), while MPI uses explicit receipt. Also, an "empty" RMI call still transmits an object identifier and a method, and requires the object and method to be looked up in a table at the receiver, while an empty IPL-level (or MPI) message requires no data or processing at all. The lower RMI array throughput on Myrinet is caused by the fact that the JVM must allocate a new array for each incoming invocation, and also zeroes (clears) this newly allocated array. The garbage collector has to be activated to reclaim the arrays. Using MPI or the low-level Ibis communication primitives, the same array can be reused for each incoming message, thus avoiding memory management overhead and also resulting in better caching behavior.

The throughputs on TCP for sending *double* values with the Sun RMI protocol are higher than the throughputs achieved by KaRMI and Ibis RMI, because the IBM class libraries use a non-standard (i.e., IBM-JIT specific) native method to convert entire double arrays to byte arrays, while the KaRMI and Ibis RMI implementations must convert the arrays using a native call per element. The latencies of KaRMI are slightly lower than the Ibis RMI latencies, but both are much better than the standard Sun RMI latency. Ibis RMI on TCP achieves similar throughput as KaRMI, but is more efficient for complex data structures, due to the generated serialization code. On Myrinet however, Ibis RMI outperforms KaRMI by a factor of 1.3–3.0 when arrays of primitive types are sent. For trees, the efficiency of the generated serialization code and effectiveness of the typed buffer scheme becomes clear, and the Ibis RMI throughput is 9.1 times higher than KaRMI's throughput.

The Manta native implementation has a better array throughput for RMI because it uses escape analysis that allows it to allocate the array parameter on a stack, thus bypassing the garbage collector. Since Ibis RMI runs on an of-the-shelf JVM, it has little control over object allocation and garbage collection, and cannot apply this optimization (of course, the JIT compilers could implement it). The throughput for a tree data structure, however, clearly is much higher for Ibis RMI than for the other implementations. On Myrinet, Ibis RMI obtains almost the full throughput offered by the IPL-level communication. This results in a throughput that is more than three times higher than the native Manta RMI implementation, clearly showing the advantages of an efficient serialization scheme (based on bytecode rewriting) combined with streaming communication. Manta RMI does not implement streaming communication to overlap serialization and communication, because Panda is not connection oriented and does not support streaming.

7.4   Conclusions and Future Work

Ibis allows highly efficient, object-based communication, combined with Java's "run everywhere" portability, making it ideally suited for high-performance computing in grids. The IPL provides a single, efficient communication mechanism using streaming and zero-copy implementations. The mechanism is flexible, because it can be configured at run time using properties. Efficient serialization can be achieved by generating serialization code in Java, thus avoiding run time type inspection, and by using special, typed buffers

to reduce type conversion and copying. Exploiting these features, Ibis is a flexible, efficient Java-based grid programming environment that provides a convenient platform for (research on) grid computing.

The Ibis strategy to achieving both performance and portability is to develop efficient solutions using standard techniques that work everywhere (for example, we generate efficient serialization code in Java), supplemented with highly optimized but non-standard solutions for increased performance in special cases. As test case, we studied an efficient RMI implementation that outperforms previous implementations, without using native code. The RMI implementation also provides efficient communication on gigabit networks like Myrinet, but then some native code is required.

In future work, we intend to investigate adaptivity and malleability for the programming models that are implemented in Ibis (i.e., GMI, RepMI, and Satin). Ibis then provides dynamic information to the grid application about the available resources, including processors and networks. For this part, we are developing a tool called TopoMon [30], which integrates topology discovery and network monitoring. With the current implementation, Ibis enables Java as a quasi-ubiquitous platform for grid computing. In combination with the grid resource information, Ibis will leverage the full potential of dynamic grid resources to their applications.

Ibis is intended as a grid programming environment, providing runtime support for grid applications, like efficient communication and subtask scheduling. An integral part of grid applications is resource management, the selection and assignment of (compute) resources to applications. Many research problems of grid resource management are currently subject to ongoing projects, like GrADS[4] or GridLab[5]. Early solutions are incorporated into grid middle-ware systems like Globus, Legion, or Harness.

Ibis will be integrated with the Globus middle-ware system that has become the de-facto standard in grids. Globus is used both for the DAS-2 system and for the GridLab project testbed (among many others). Ibis will request compute resources via the Globus system. The GridLab project is currently developing a grid resource broker that will be able to map resource requests to well-suited machines; Ibis will also integrate such a brokering tool in conjunction with the Globus platform. Monitoring data about grid network connectivity (for optimizing communication between grid sites) will also be retrieved via grid information services, like Globus' GIIS. In general, we will adhere to the recommendations of the Global Grid Forum, such as the Grid Monitoring Architecture (GMA).

---

[4]http://www.hipersoft.rice.edu/grads/
[5]http://www.gridlab.org/

Chapter 8

Grid-Enabled Satin

```
Benefits should be conferred gradually;
and in that way they will taste better.
```

```
- Niccolo Machiavelli
```

Chapters 5 and 6 described the Satin design and implementation, and demonstrated that divide-and-conquer applications can indeed be efficiently executed in a grid environment. However, since the original Satin implementation, Java virtual machines have become much more efficient, bringing a high performance implementation of Satin in pure Java within reach. In the previous chapters, we have identified and eliminated many bottlenecks in Java serialization and communication, and also in Satin execution speed and load balancing. Also, some functionality (e.g., exception handling for spawned methods) was not defined in the Satin prototype that has been described in the previous chapters. Moreover, Satin lacks some desirable features (support for speculative parallelism) that are present in other divide-and-conquer systems.

We now want to apply the lessons we learned from the native Manta implementation to a new, pure-Java version of Satin, with extended functionality. Our goal is to make the new version truly usable in a grid setting. To summarize, the Satin implementation presented in the previous chapters still has five problems:

1. there is no well-defined way of handling exceptions thrown by spawned work (see Section 5.1);

2. there is no mechanism to execute a user-defined action when a spawned method is finished (this is useful for speculative parallelism);

3. there is no mechanism to retract work that was speculatively spawned;

4. deployment in a heterogeneous grid environment is difficult due to portability problems, because Satin is implemented in a native Java compiler (Manta), and uses a native runtime system.

5. Satin has a closed-world assumption, because it uses Panda for low-level commu-
   nication, and Panda does not allow the adding and removing of machines during a
   parallel run.

In this chapter, we attempt to solve these five problems. Handling of exceptions
(item 1) should be done on the machine that spawned the work, as is done with RMI,
for instance. It is also desirable to provide a mechanism that allows user code to be ex-
ecuted when spawned work is finished (item 2). This action should also be triggered on
the machine that spawned the work, because the data (e.g., local variables) that is needed
to execute the handler resides there. This chapter gives one novel solution that solves
both problems at the same time. Furthermore, we describe the implementation of an abort
mechanism that can be used to retract speculatively-spawned work (item 3). Moreover,
we solve the portability and closed-world problems (items 4 and 5), by reimplementing
Satin in pure Java, on top of Ibis.

This chapter describes the design, implementation and performance of Satin on top
of Ibis. We will call the new version Satin/Ibis. The new version has more functionality
than the Satin version that is integrated into Manta, which we will call Satin/Manta in
this chapter to avoid confusion. Satin/Ibis supports exception handling, allows actions
to be triggered when spawned work is finished, and features a distributed abort mecha-
nism to retract speculative work. With these features, Satin/Ibis allows a larger class of
applications to be expressed than Satin/Manta. Example algorithms that need the mech-
anisms introduced in this chapter, and that can be expressed with Satin/Ibis, but not with
Satin/Manta, are algorithms that use AND/OR parallelism, MTD(f) [133], and Nega-
Scout [138].

Now that we have shown (in Chapter 6) that Satin can run efficiently on wide-area
systems, we want to provide a Satin implementation that is written in pure Java, and uses
Ibis for communication between JVMs. The result is a divide-and-conquer platform that
supports malleability (machines can be added and removed during the computation), and
that can run in any heterogeneous environment (i.e., the grid), without any porting effort.
The only requirement to run Satin/Ibis is a JVM. We show that, even with these require-
ments, divide-and-conquer primitives can be implemented efficiently with compiler and
runtime system support. The Satin/Ibis compiler, called *satinc*, can be used to generate
parallel bytecode for Satin programs. The generated parallel bytecode can be executed on
any JVM.

The contributions of this chapter are the following:

1. we describe Satin/Ibis, a Java-centric divide-and-conquer system that can run effi-
   ciently on the grid without any porting effort;

2. we give a classification of abort mechanisms;

3. we show that Java's exceptions can be used to implement a mechanism that executes
   a user-defined handler when spawned work is finished;

4. we describe the design and implementation of a mechanism that can retract (spec-
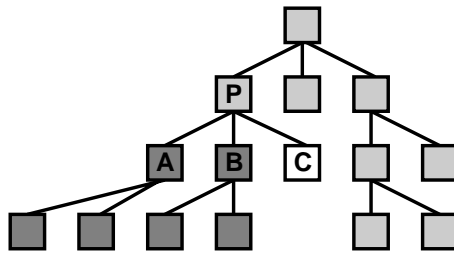   ulatively) spawned computations on a distributed-memory architecture.

Figure 8.1: A search tree where work can be aborted.

5. we show that it is possible to implement these mechanisms without altering the Java language;

6. we demonstrate that Satin can be implemented efficiently with compiler and run-time system support, even in pure Java.

7. we verify the simulation results that are presented in Chapter 6, by running Ibis and Satin with RS and CRS on a *real* wide-area testbed.

8. we describe hands-on experience and demonstrates that, in practice, the Java-centric approach to grid computing greatly simplifies running parallel applications on a heterogeneous system.

The remainder of this chapter is structured as follows. First, we give a brief introduction of existing mechanisms to abort speculative work. Next, in Section 8.2, we describe the design of a new mechanism to execute a user-defined action when spawned work is finished. Also, we describe the abort mechanism that we implemented for Satin/Ibis. Section 8.3 deals with the implementation of the new Satin version in Java, on top of Ibis. The performance of Satin/Ibis is evaluated in Section 8.4, both using micro benchmarks and applications. A case study of an application (Awari) that uses Satin's abort mechanism to implement speculative parallelism is presented in Section 8.6. We discuss related work in Section 8.7. Finally, we draw our conclusions in Section 8.8.

8.1 Introduction to Inlets and Abort Mechanisms

In many parallel divide-and conquer applications, such as algorithms that use AND/OR parallelism and game-tree search, it is desirable to have a mechanism to abort useless computations. When searching a game tree in parallel, for instance, speculatively-spawned work may become unnecessary due to new intermediate results. Many heuristic game-tree search algorithms speculatively execute work. When, during the computation, it becomes clear that the work that was speculatively started can never give a better solution than the results that are found so far, it is desirable to abort the speculatively-started work, and do useful work instead. This is shown in Figure 8.1. The nodes *A*, *B* and *C* are expanded in

```
 1  cilk int foo(int i) {
 2      do_intermediate_amount_of_work();
 3      return i;
 4  }
 5
 6  cilk int cilk_main(int argc, char *argv[]) {
 7      int i, nrFinished = 0;
 8
 9      inlet void catch(int res) {
10          printf("result of foo was %d\n", res);
11          if(++nrFinished == 2) abort;
12      }
13
14      for(i=0; i<3; i++) {
15          catch(spawn foo(i));
16      }
17      sync;
18  }
```

Figure 8.2: *Two out of three*: an example of an inlet and abort in Cilk.

parallel. The expanding of the white node (*C*) resulted in a good solution. This solution is returned to the parent node (*P*), which can then abort its other children which have become irrelevant (the dark nodes *A* and *B*). Also, all nodes below the irrelevant children are aborted.

Cilk [22], a divide-and-conquer system that extends the C language, provides a combination of *inlets* and an explicit *abort* operation that allows the application to retract useless computations. An inlet is a piece of code that is executed in the scope of the parent when a spawned computation is finished. This piece of code can access the variables in the parent's stack frame. This way, the inlet can, for instance, be used to update the search window in the parent frame. The inlet can also issue an *abort* operation, killing all other children that were spawned by the parent. It is important that inlets are executed as soon as possible, because an inlet might contain an abort operation that can prune a large amount of work.

The way in which work can be aborted depends on the memory model that is provided by the programming environment. If shared memory is offered, an abort can be implemented by the programmer, as child nodes can have pointers to their parent nodes, and can thus set an abort flag or change search parameters in their parents. Likewise, the parent node may have pointers to the children it generated, which allows the parent to set aborts flags, or to change search parameters in the children. An explicit abort mechanism is still useful, however, because this alleviates the need for polling abort flags, thus improving performance. Furthermore, this way, work can be aborted at any point, also when the computation is not polling abort flags. Moreover, an explicit abort mechanism is more programmer friendly, as parent and child pointer administration is no longer needed. An alternative way to implement an abort mechanism on a shared-memory architecture is to use the *setjmp/longjmp* mechanism.

When the programming environment does not provide shared memory, explicit abort mechanisms are required to abort speculative work that has become useless. The abort mechanism is implemented in the runtime system of the programming environment. If a parent executes an abort, all unfinished work that was spawned by it will be retracted.

category	abort type	implementation	
		shared memory	distributed memory
1	all work	global variables	replicated objects / inlets & abort
2	spawned children (subtrees)	child pointers	inlets & abort
3	dynamically changing parameters		
A	- global parameters	global variables	replicated objects
B	- parameters of spawned children	child pointers	child parameter updates

Table 8.1: Classification of Abort Mechanisms.

Figure 8.2 shows an example of the use of inlets and abort in Cilk. The inlet is declared inside the main function, something that is normally not possible for functions in C. The inlet code can access variables in the frame of its parent, in this case *nrFinished* is increased by one. When *nrFinished* reaches the value two, the third spawned *foo* function is aborted. Satin/Ibis implements a similar inlet mechanism and explicit abort primitive, but without the need for language extensions.

8.1.1   A Classification of Abort Mechanisms

Depending on the application and memory model of the programming environment, different abort mechanisms are needed. Table 8.1 gives a classification of abort mechanisms. We differentiate between shared and distributed memory, because the memory architecture influences the implementations of the different abort mechanisms. In our classification, we treat DSM systems as shared memory. It is likely however, that using a DSM (i.e., a general purpose solution) to implement aborts on a distributed-memory machine is less efficient than using a distributed abort mechanism.

Category 1

The first category, *abort all work*, is the most straightforward mechanism. All work is aborted when some *global* condition becomes true. For instance, with single-agent search, when a solution is found, all other work can be aborted. On a shared-memory system, this can be easily implemented using a global variable. The worker threads poll the global condition, and stop when it becomes true. When shared memory is not available, a replicated object can be used to simulate this behavior, or a simple global abort can be used to stop all computation. A global abort is potentially more efficient than the use of replicated objects, because the latter requires constant polling of the (replicated) condition, whereas the global abort interrupts the computation.

Examples of algorithms that need global aborts are single-agent search, such as the 15-puzzle and N-queens. For performance measurements in the previous chapters we used versions of these algorithms without aborts (i.e., all spawned work is also executed), which makes these programs deterministic (see Sections 2.7.7, 2.7.8 and 2.7.17).

Category 2

The second category is more difficult because some or all spawned children of a given node (subtrees in the computation) must be aborted, depending on some *local* condition. Other parts of the spawn tree are not affected. AND/OR-parallelism, for example, needs this type of abort: when a child of an AND node returns *false*, the remaining children can be aborted. An example abort of this category is shown in Figure 8.1.

When shared memory is available, this abort mechanism can be implemented by traversing parent and child pointers that connect nodes in the spawn tree, although this is not very programmer friendly, and is therefore better done by a runtime system. In a distributed-memory setting, the manual approach can again be simulated with replicated objects, but this would require the creation of a replicated object *per node* in the search tree. The creation of a replicated object is an expensive operation. Thus, using replicated objects for this abort category is undesirable. Instead, on shared-memory systems, but especially on distributed-memory systems, a separate abort mechanism is needed to dispose of speculatively-spawned work. When children have to be conditionally aborted, as in the "two out of three" example (see Figure 8.2), an inlet mechanism is also needed. Algorithms that need this type of aborts are AND/OR parallelism, MTD(f) [133] and NegaScout [138].

Category 3

The third category is the most complex one. The parallel computation is steered by dynamically updating parameters (e.g., a global search bound, or a local search window). Due to the changing parameters, the computation conditionally aborts work that was already spawned. This category can be subdivided into two parts: the parameters may be either global or local.

For *global* parameters, global variables can be used to store the parameter values. An example of an algorithm with a global search parameter is TSP, which uses one global minimum path length in the whole computation. When a spawned thread finds that the partly-calculated route is longer than the global minimum, the current work can be pruned. When a solution is found that is shorter than the current minimum path, the global minimum is updated. The other spawned methods will now use the new sharper bound to prune work. This type of abort is not possible on pure distributed-memory systems. When available, replicated objects can be used to implement this type of abort on distributed-memory systems (see Section 5.2.5).

Some algorithms need to abort work depending on *local* parameters. An example is the Alpha-Beta algorithm [91], which uses a search window, (α, β). The search window is used to specify the range within which results are interesting. Search results outside the range are irrelevant and can be aborted. A search window is kept *per node* in the search tree. A child node's α and β values can be dynamically updated when search results become available in its parent.

On shared memory systems, this abort mechanism can be implemented by traversing the pointers that connect the spawn tree. On distributed memory however, this is more difficult. An abort primitive as shown in Figure 8.2 is not sufficient, as there is no way

to change parameters in the spawned children. The children can change parameters of their parents, using inlets, but the other way around is not possible. What is needed is a mechanism that lets the parent send update messages to the children it spawned.

Summary

Table 8.1 shows that offering spawn and sync primitives as well as replicated objects (as Satin/Manta did) is not sufficient to *efficiently* express all algorithms that speculatively spawn work. Category 1 can be expressed more efficiently and easily with inlets and an abort mechanism. Moreover, some algorithms need to retract entire subtrees of speculative work (i.e., category 2), which is even harder to implement (efficiently) without inlets and an abort mechanism. Satin/Ibis provides inlets and an explicit abort mechanism which can be used together to implement aborts of the categories 1 and 2. For category 3A, shared variables or replicated objects are needed. Satin/Ibis uses RepMI to implement this. Neither Cilk nor Satin/Ibis implement category 3B, and, as far as we know, neither does any other distributed language. Multigame, a problem solving environment for distributed game-tree search, does implement aborts of category 3B.

8.2   Design of Inlets in Satin/Ibis

In this section we will explain the design of the inlet and abort mechanisms in Satin/Ibis. A prerequisite is that we do not modify or extend the Java language itself.

An inlet is a piece of code that is executed when a spawned method is finished. An inlet must run in the context of the spawner, because it should be able to access its local variables. In Cilk, inlets are an extension of the C language. Syntactically, they are programmed as a *nested function* inside the spawner (see Figure 8.2). Subroutines are not defined in the C language. We want a clean integration of inlets into Java, which also does not have nested functions, so another mechanism is required. We also have to define meaningful exception semantics for spawned methods. The key idea is to combine the two, and use the exception mechanism of Java to implement inlets.

Figure 8.3 shows the "two out of three" example shown earlier in Figure 8.2, but now in Satin/Ibis. The method *foo* in the class *TwoOutOfThree* will be spawned (line 24), because it is tagged in the interface *TwoOutOfThreeInter* (line 9). In the Cilk example, *foo* returned an *int* result. The Satin/Ibis version returns its result via an exception of the type *Done*. This makes it possible to use the catch block around the spawn of *foo* (lines 25–29) as an inlet. In Figure 8.3, *foo* gets spawned three times. When a spawned method is finished, it will return with an exception, and the code in the catch block will run. The inlet can access all local variables, and can thus increment and test the counter *nrFinished*. When the counter reaches two, the outstanding work which is not finished yet is aborted using the *abort* method (line 27) which is, like the *sync* method, inherited from *ibis.satin.SatinObject*. The precise semantics of the program will be described in more detail in the following sections.

Satin/Ibis provides a class *Inlet*, which is a subclass of *Throwable* (the base class of all exceptions), that can be extended instead of the standard *Exception* class. In the example,

```
 1  import.ibis.satin.*;
 2
 3  class Done extends Exception {
 4      int res;
 5      Done(int res) { this.res = res; }
 6  }
 7
 8  interface TwoOutOfThreeInter extends Spawnable {
 9      void foo(int i) throws Done;
10  }
11
12  class TwoOutOfThree extends SatinObject implements TwoOutOfThreeInter {
13      void foo(int i) throws Done {
14          do_intermediate_amount_of_work();
15          throw new Done(i);
16      }
17
18      public static void main(String[] args) {
19          TwoOutOfThree t = new TwoOutOfThree();
20          int nrFinished = 0;
21
22          for(int i=0; i<3; i++) {
23              try {
24                  t.foo(i); // Spawn.
25              } catch (Done d) {
26                  System.out.println("foo res: " + d.res);
27                  if (++nrFinished == 2) t.abort();
28                  return; // Exit the inlet, do not fall through.
29              }
30          }
31          t.sync();
32      }
33  }
```

Figure 8.3: *Two out of three*: an example of an inlet and abort in Satin/Ibis.

this would mean that class *Done* would extend the *Inlet* class. The use of the *Inlet* class is not obligatory, but is more efficient, because the *Inlet* class does not contain a stack trace (a trace of the call stack leading to the exception, which is useful for debugging). We found that constructing the stack trace is an expensive operation (see Section 8.4.1), and a stack trace is not useful when an exception is thrown to trigger an inlet. To circumvent the expensive stack trace creation, the *Inlet* class overrides the *fillInStackTrace* method of the *Throwable* class with an empty method.

A problem is the context in which inlets run. They should run in the frame of the spawner, as local variables of the spawner can be accessed inside the catch block. However, as multiple parallel threads may be spawned and may throw exceptions at different times, several inlets may be executed in a single method, as is the case in the "two out of three" example. The flow of control in the inlet can be complex. The execution can even leave the catch block, by means of a *break* or *continue* statement for example. Without

restricting the code that is allowed in the catch blocks, inlets can run indefinitely long, and can jump to any instruction in the method. Therefore, we chose to conceptually spawn a new thread for each inlet. This thread is different from normal Satin/Ibis threads (and normal Java threads), as it *shares* its local variables and parameters with the spawner thread. Satin/Ibis guarantees that no two inlets of the same spawner will run at the same time, and that an inlet does not run concurrently with its spawner. This way, no locks are needed to protect access to local variables.

Our approach circumvents many of the well-known problems [32, 115] that arise when asynchronous exceptions are used. Because a new thread is created to handle the exception, the spawner of the work does not have to be interrupted. Systems that support asynchronous exceptions also have to support critical regions [32, 115] to avoid the interruption of a thread at an inconvenient time. The programmer then has to annotate the code to indicate what the critical sections are. Satin/Ibis avoids these problems by creating a new thread to handle the exception, and by guaranteeing that the inlets do not run concurrently with the spawner. Still, our mechanism is as expressive as the traditional approach of interrupting the spawner, because the new thread can read and write the local variables and parameters of the spawner.

Because (conceptually) a thread is started for each inlet, the semantics of inlets are different from the exception semantics in standard Java. It is therefore, in general, not correct to run code with inlets sequentially on a JVM, without compiling it with *satinc* first. This can be seen in Figure 8.3, where there is a *return* statement in the catch block, to avoid that the inlet thread falls through.

Satin/Ibis's exception mechanism is different from the RMI exception mechanism, which also offers remote exceptions (see Section 2.3.3). The most important difference is the fact that RMIs are synchronous, while spawn operations are asynchronous. One spawner thread can spawn many method invocations before a sync operation is reached. Exceptions thrown by spawned methods can arrive at any time, and should be handled as soon as possible to avoid search overhead. Therefore, they should be executed asynchronously. The difference between RMI exception handling and Satin/Ibis exception handling is shown in an example of the possible behavior of a Satin/Ibis program in Figure 8.4. As can be seen in the figure, RMIs are completely synchronous, also in the case of exceptions. With Satin/Ibis however, the spawner continues when a task is spawned. Exceptions are handled asynchronously by a new thread that shares its local variables and parameters with the spawner, but not by the spawner itself.

With our Java language design, it is even possible to have different inlets for spawns in one method. This can be programmed in two ways, both of which are shown in Figure 8.5. The interface *SpawnerInter* contains the methods *foo* and *faa*, and is omitted for brevity. In *method1*, the spawns are in the same try block, but as the spawned methods return with different exception types, the spawner can distinguish them by providing multiple catch clauses. In *method2*, the different spawns have a different try-catch-block associated with them. Both mechanisms are allowed in Satin/Ibis.

It also is possible to (re)throw an exception from inside an inlet. Satin/Ibis treats this as an implicit abort. Thus, all children of the spawner are aborted, and the exception triggers a new inlet in the parent of the spawner. This way, inlets can recursively unwind the stack. When an exception thrown by a spawned method is *not* caught, this is also regarded as an
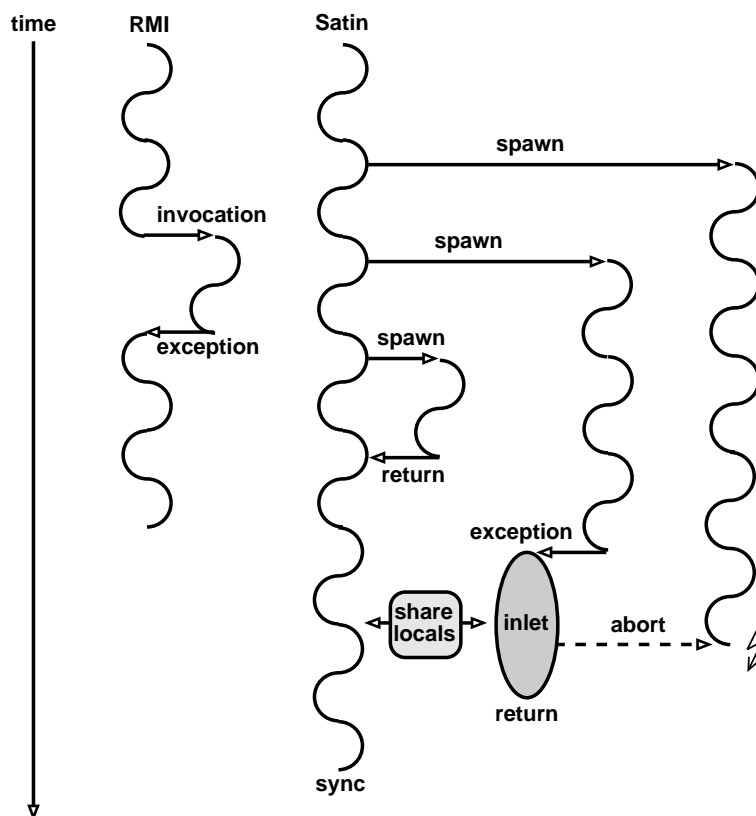
Figure 8.4: Exceptions in RMI and in Satin/Ibis.

implicit abort. Next, the exception is rethrown by the Satin runtime system, and can be caught by the parent of the spawner.

8.2.1   Design of the Abort Mechanism in Satin/Ibis

The abort primitive is used, mostly in combination with inlets, to abort speculative work that has become unnecessary. An example of the use of an explicit abort primitive is shown in Figure 8.3 (line 27). In the example, three threads are speculatively spawned. When any two threads have finished, the third thread is aborted. As stated in the previous section, inlets run in the context of the spawner, and may thus abort threads that were started by the spawner.

   In the Satin/Ibis language we chose to use an explicit abort mechanism. An implicit abort scheme would also have been possible. In that case, when a spawned method throws an exception, the exception is forwarded to the spawner, which then automatically kills all outstanding work that is spawned. After the work is aborted, the code in the catch-

```
1  import ibis.satin.*;
2
3  class Spawner extends SatinObject implements SpawnerInter {
4      void foo() throws FooException() {
5          // foo code
6      }
7
8      void faa() throws FaaException() {
9          // faa code
10     }
11
12     void method1() {
13         try {
14             foo(); // Spawn.
15             faa(); // Spawn.
16         } catch (FooException e1) {
17             // Inlet code for foo.
18         } catch (FaaException e2) {
19             // Inlet code for faa.
20         }
21         sync();
22     }
23
24     void method2() {
25         try {
26             foo(); // Spawn.
27         } catch (Exception e) {
28             // Inlet code for foo.
29         }
30         try {
31             faa(); // Spawn.
32         } catch (Exception e) {
33             // Inlet code for faa.
34         }
35         sync();
36     }
37 }
```

Figure 8.5: Multiple inlets in one method.

block is executed. This scheme has the advantage that it is somewhat simpler than an
explicit abort. The catch-block does not have to be a new thread, as only one exception
can be thrown, because an exception results in the aborting of the other children. The
programmer has to think in a new way about aborts, however, as it is the child that tells
the parent to abort its remaining children, whereas with the traditional, explicit abort
it is the parent who makes the abort decision. Another disadvantage of implicit abort
is that it is not possible to express the "two out of three" example with it: as soon as
one spawned method returns with an exception, the others are always killed. The less
intuitive semantics and the weaker expressiveness together made us opt for the explicit
abort scheme in Satin/Ibis.

Satin/Ibis guarantees that no results of outstanding spawned method invocations will be assigned after an abort operation, neither will any further inlets that are triggered by outstanding jobs be executed. In Section 8.3.5, we will demonstrate that the abort mechanism can be implemented efficiently and in an asynchronous fashion, even with these strict semantics.

8.3   Implementation of Satin/Ibis

Satin/Ibis cleanly integrates into Java, and uses no language extensions. The fact that Satin/Ibis does not modify the Java language has several practical advantages. Satin/Ibis programs can be compiled by any Java compiler. When Satin/Ibis applications need to run in parallel, the Satin/Ibis compiler, *satinc*, which is also written in Java, is used to convert the sequential *bytecode* to a parallel version. Because Satin programs can be compiled with any Java compiler, satinc can completely operate on the bytecode level. It does not need to parse the Java source, which is much more difficult. Satinc uses the Jikes Bytecode Toolkit [96] (unrelated to the Jikes Java compiler and the Jikes RVM just-in-time compiler) to implement bytecode rewriting.

The Satin/Ibis compiler reads the Java bytecode files, and recognizes the special "marker" interfaces. It then rewrites the normal method invocations to spawned methods invocations, using calls to the Satin/Ibis runtime system, which is also written in Java. The output of the Satin/Ibis compiler is again Java bytecode which can run in parallel on any JVM (or multiple JVMs). In short, the only requirement to run Satin programs is a JVM, because all parts of the system (even the middle-ware layer) are written in Java. This way, Satin/Ibis exploits the "write once, run everywhere" feature of Java. The challenge that comes with this approach is to achieve good performance, because Java neither allows pointer arithmetic nor pointers to the stack, which are heavily used by traditional divide-and-conquer implementations, such as Cilk and Satin/Manta. In this section, we will describe our implementation, and the optimizations we had to implement to make Satin/Ibis efficient.

8.3.1   Spawn and Sync

The implementation of the *spawn* and *sync* primitives in Satin/Ibis is largely the same as the implementation for Satin/Manta, as described in Chapter 5 (Section 5.2). However, the use of pure Java makes the Satin implementation on top of Ibis more difficult, because Java does not allow pointers to the stack. Satin/Ibis also implements the important *serialization on demand* optimization (see Section 5.2.2), and uses similar invocation records to do this. Java's serialization mechanism (or the optimized serialization protocol provided by Ibis) is used in Satin/Ibis for marshalling the parameters to a spawned method invocation when work is stolen. In the local case, no serialization is used, which is of critical importance for the overall performance. The Satin/Ibis implementation avoids thread creation altogether which has a large positive impact on performance.

Figure 8.6 shows the parallel code that is generated by the Satin/Ibis compiler for the *Fib* example from Figure 5.1 (see Chapter 5). Although *satinc* produces bytecode,

```
 1 public void long fib(long n) {
 2     long x, y;
 3     if(n < 2) return n;
 4     SpawnCounter spawnCounter = Satin.getSpawnCounter();
 5     InvocationRecord outstandingSpawnsList = null;
 6
 7     outstandingSpawnsList =
 8         getInvRec_Fib_fib(n−1,
 9         spawnCounter, outstandingSpawnsList,
10         0 /* storeId for x */, 0 /* spawnId for first spawn */);
11     Satin.spawn(outstandingSpawnsList);
12
13     outstandingSpawnsList =
14         getInvRec_Fib_fib(n−2,
15         spawnCounter, outstandingSpawnsList,
16         1 /* storeId for y */, 1 /* spawnId of second spawn */);
17     Satin.spawn(outstandingSpawnsList);
18     Satin.sync(spawnCounter);
19     while(outstandingSpawnsList != null) {
20         InvocationRecord curr = outstandingSpawnsList;
21         outstandingSpawnsList = outstandingSpawnsList.next;
22
23         switch(curr.storeId) {
24             case 0: x = curr.result; break;
25             case 1: y = curr.result; break;
26         }
27         deleteInvocationRecord_Fib_fib_foo(curr);
28     }
29     deleteSpawnCounter(spawnCounter);
30     return x + y;
31 }
```

Figure 8.6: Pseudo Code generated by the Satin/Ibis compiler for the *fib* method.

Figure 8.6 shows pseudo code to enhance readability. The gray boxes denote code that directly relates to the original *Fib* example. We will now explain the generated code in more detail.

As described in Section 5.2.1, Satin/Ibis needs a per-method counter for methods which execute spawn operations. This counter is called the *spawnCounter*, and counts the number of pending spawns that have to be finished before the method can return. The Satin/Ibis compiler generates code to allocate a spawn counter *object* (line 4) at the beginning of each method that executes spawn operations. A reference to the spawn counter is also stored in the invocation record. This way, the counter can be decreased when the results for a spawned method become available. In Satin/Manta, the spawn counter was a normal integer value *on the stack*, and a pointer to this value was stored in the invocation record. This is not possible in pure Java, as pointers to the stack cannot

be created. Also, all methods that spawn work are rewritten by the Satin/Ibis compiler to keep a list of invocation records, called *outstandingSpawnsList*, which represents the outstanding work that was spawned but is not yet finished.

For efficiency, a pool of both unused invocation records and *spawnCounter* objects is cached by the Satin/Ibis runtime system, thus reducing the need for creating objects on the critical path. Also, because the Satin/Ibis runtime system is entirely single threaded, there is no need for locks to protect the caches. On an SMP, there is one thread per processor, and each thread has a private cache, again to avoid locking.

There is another important optimization that is implemented by the Satin/Ibis compiler regarding the *spawnCounter*. Instead of allocating a new *spawnCounter* object at the beginning of methods that can be spawned, *satinc* tries to delay the allocation as long as possible. Of course, the *spawnCounter* has to be allocated before a spawn operation is reached, and special care has to be taken that the flow of control does not jump over the allocation. The rationale behind the optimization is that recursive programs often have a *stop condition* located at the beginning of the method. For the *Fibonacci* example, for instance, the stop condition "if(n < 2) return n" is in the first statement of the method. If there is no spawn operation before the stop condition, it is better to insert the *spawnCounter* allocation *after* the evaluation of the condition, thus avoiding the unnecessary allocation and deallocation of the spawnCounter when the stop condition evaluates to *true*. This is reflected in Figure 8.6, where the allocation is inserted at line 4, *after* the stop condition on line 3. This way, leaf nodes in the spawn tree do not pay the overhead of allocating the *spawnCounter*.

When a program executes a spawned method invocation, Satin/Ibis redirects the method call to a stub. This stub creates an invocation record (lines 7–10 and 13–16), which describes the method to be invoked, the parameters that are passed to the method, and a handle to where the method's return value has to be stored, called the *storeId*. The invocation records are method specific, and are generated by the Satin/Ibis compiler. This way, no runtime type inspection is required. For primitive types, the value of the parameter is copied. For reference types (objects, arrays, interfaces), only a reference is stored in the record. From an invocation record, the original call can be executed by pushing the values of the parameters (which were stored in the record) onto the stack, and by calling the Java method. This can be done both for local and stolen jobs. The *spawnId* is used to implement the inlet mechanism, and is unused in the Fib example. The generated code for inlets will be discussed in Section 8.3.4.

When an invocation record is created, it is added to the *outstandingSpawnsList* (lines 7 and 13). Next, generated code calls the Satin/Ibis runtime system to put the invocation record in the work queue (lines 11 and 17). The runtime system also puts a unique stamp on each job, which is later used to identify it. The stamp of the parent is stored in the invocation record as well. The latter is used to implement the abort primitive.

The sync operation is rewritten to a call to the Satin/Ibis runtime system (line 18) which executes work from the job queue until the *spawnCounter* reaches zero. After the sync operation, code is inserted to traverse the *outstandingSpawnsList* (lines 19–28), and to assign the results of the spawned methods out of the invocation records to the destination variables (which may be on the stack of the method). In Satin/Manta, the invocation records stored pointers to the result variables, which could then be assigned

when the spawned methods were finished. This approach is not possible in Satin/Ibis, again because pointers to the stack are prohibited in Java.

8.3.2 Load Balancing

The load-balancing mechanisms of Satin/Ibis are identical to those described earlier (in Chapter 6). However, some extra administration is needed to implement inlets and the abort mechanism. In Satin/Ibis, one important invariant always holds. All invocation records in use (jobs that are spawned but not yet finished) are in one of three lists:

- The *onStack* list, for jobs that are on the Java stack, because they are currently being worked on by the local processor.

- The *work queue*, for work that is spawned, but not yet running.

- The *outstandingJobs* list, for stolen work.

This way, the Satin/Ibis runtime system can always find information about any job that was spawned, but is not yet finished.

The invocation records that describe the spawned method invocations are stored in a double-ended job queue when the work is spawned. When a job is taken from the queue to be executed locally, it is inserted in the *onStack* list, to indicate that work on this job is in progress. When the job is finished, it is removed from the *onStack* list again. When a job is stolen, the victim moves the job from the work queue to the *outstandingJobs* list. Next, the job is sent over the network to the thief.

When a stolen job is finished, the return value and the job's original stamp (see Section 8.3.1) will be serialized and sent back to the originating node (i.e., the victim). There, the original invocation record can be found on the *outstandingJobs* list using the job's stamp. Next, the spawn counter (that can be found via the invocation record) is decreased, indicating that there is one pending spawn less.

8.3.3 Malleability

Satin/Manta has a closed-world assumption, because it uses Panda for low-level communication, and Panda does not allow the adding and removing of machines during a parallel run (i.e., it does not support malleability). Ibis, however, does support malleability (see Section 7.1). Implementing malleability in Satin/Ibis is straightforward. When a machine joins the running computation, Ibis ensures that all participants get an upcall with a contact point for the new machine as a parameter. Using this contact point, Satin/Ibis sets up a communication channel with the new participant. From this moment on, the new machine can steal work over the new communication channel, and vice-versa.

Currently, a machine is only allowed to leave a running computation when it has neither work in its queue nor on its execution stack (i.e., it is idle). When a machine wants to leave the computation, all other participants will get an upcall again, so all communication channels can be closed, and the leaving machine can be removed from Satin's administration.

```
1  final class TwoOutThree_main_LocalRecord {
2      Done d;           // Locals of the main method
3      int nrFinished; // that are used from inside
4      TwoOutThree t;  // an inlet.
5  }
```

Figure 8.7: The generated localrecord for the main method of *Two out of three*.

```
1  public static void main(String[] args) {
2      TwoOutThree_main_LocalRecord locals =
3        new TwoOutThree_main_LocalRecord();
4      locals.t = new TwoOutOfThree();
5      locals.nrFinished = 0;
6
7      for(int i=0; i<3; i++) {
8          try {
9              locals.t.foo(i); // spawn
10         } catch (Done d) {
11             System.out.println("foo res: " + d.res);
12             if (++locals.nrFinished == 2) locals.t.abort();
13             return;
14         }
15     }
16     locals.t.sync();
17 }
```
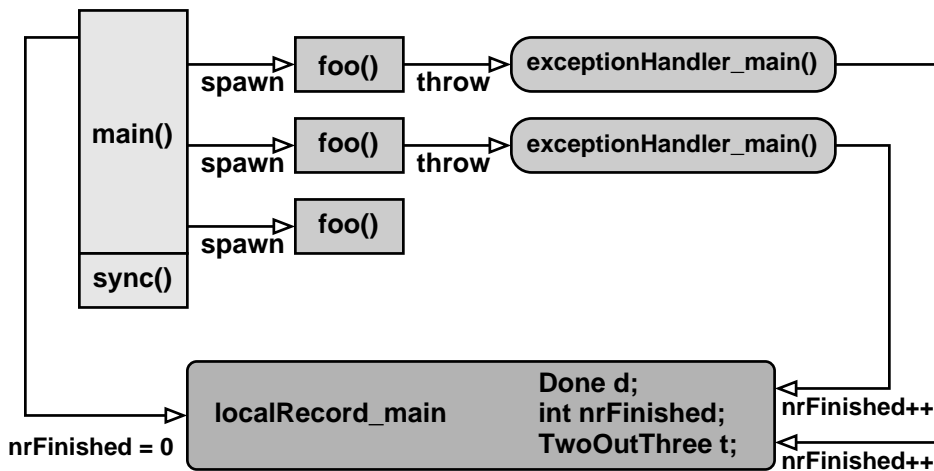
Figure 8.8: The rewritten version of the main method of *Two out of three (pseudo code)*.

8.3.4   Inlets

When a method contains a try-catch-block around a spawned method invocation (i.e., an inlet), the Satin/Ibis compiler must take special measures. The use of Java complicates things. In Cilk or a native Java system, pointers to the stack can be used to access the local variables in the frame of the parent. This is impossible in Java without modifying the JVM, which we clearly do not want, for portability reasons.

Because Java does not allow pointers to stack and inlets have to share local variables with the method that spawned them, locals must be handled in a special way. The solution is to allocate a special *localrecord* that contains a field of the same type for each local variable of the method that is used by the inlet. The code of the method is then rewritten to use the fields in the localrecord instead of the local variables. Instead of directly sharing the local variables, the inlet and the spawner now share a reference to the localrecord, and use an indirection to read and update the values. Parameters to the method are treated in the same way as local variables. The Satin/Ibis runtime system is entirely single-threaded, so there is always exactly one thread that can access the localrecord at a given time. Therefore, no locks are needed to protect the localrecord. Figure 8.7 shows the localrecord that is generated by the Satin/Ibis compiler for the "two out of three" example in Figure 8.3. In reality, the name mangling is more complex, but this is omitted for readability. Also, localrecords are cached in our implementation (not shown).

Figure 8.9: Execution of *two out of three*.

Pseudo code for the rewritten *main* method is shown in Figure 8.8. Of course, the extra indirection that is used to access local variables comes with some overhead. This overhead can be reduced in some cases, by only rewriting variables that can be used by an inlet. This requires some control-flow analysis, as the flow of control may leave the catch block. A common case, however, is an inlet that conditionally executes an abort, and then finishes (i.e., returns). In this case, it is straightforward to find out which variables must be rewritten. Satin/Ibis implements this optimization, which is reflected in the example: the localrecord for *main* (see Figure 8.7) does not contain the loop counter *i* and the parameter *args*, because neither is used inside the inlet. Also, in Figure 8.8, the loop counter *i* is accessed directly (without using an indirection via the localrecord).

When a spawned method throws an exception, the code for the inlet must be executed. To do this, the Satin/Ibis compiler generates a special *exception handling clone* for all methods that contain an inlet. This clone contains all code of the original method, plus an extra preamble that jumps to the correct catch block depending on the type of the thrown exception and the location of the spawn. The clone must contain all code, and not just the catch block, because the flow of control may leave the catch block due to a *break* or *continue* statement, for instance. Furthermore, the execution can fall through the end of the catch block, just as it would happen in sequential Java.

The structure of the inlet mechanism for the "two out of three" example is shown in Figure 8.9. We will now describe the example in more detail. The Satin/Ibis compiler has rewritten *main* (because it contains an inlet) to use a localrecord instead of local variables on the stack. As main runs, the variable *nrFinished*, which now resides in the localrecord, is initialized to zero, and three spawn operations are executed. When the sync operation is encountered, *main* blocks, and the Satin/Ibis runtime system starts running work out of the queue. In this case, one of the spawned *foo* methods. The first *foo* that finishes throws an exception, which is intercepted by the Satin/Ibis runtime system.

```
1  public static void exceptionHandlingClone_main(int spawnId,
2    TwoOutThree_main_LocalRecord locals, Throwable exception,
3    InvocationRecord thrower) {
4
5      switch (spawnId) { // Cast the exception to the correct type.
6          case 0:
7              if(exception instanceof Done) {
8                  Done d = (Done) exception;
9                  push(d); // Push exception on the stack.
10                 goto inlet1; // Jump to catch block.
11             }
12         break;
13     }
14
15     locals.t = new TwoOutOfThree();
16     locals.nrFinished = 0;
17
18     for(int i=0; i<3; i++) {
19         try {
20             locals.t.foo(i); // Spawn.
21         } catch (Done d) {
22 inlet1:
23             System.out.println("foo res: " + d.res);
24             if (++locals.nrFinished == 2) locals.t.abort();
25             return;
26         }
27     }
28     locals.t.sync();
29 }
```

Figure 8.10: The generated exception handling clone for the main method of *Two out of three* (pseudo code).

The exception, which triggers the inlet inside main, is passed as a parameter to *the exception handling clone of main*, which was generated by the Satin/Ibis compiler. The exception handler looks at the location where *foo* was spawned, using the *spawnId* (see also Figure 8.6, line 10 and 16) and chooses the correct inlet, also depending on the type of the exception. In this case, there is only one inlet. Therefore, the handler jumps to the catch block that handles the exception. Next, the counter *nrFinished* is incremented (via the localrecord) and the inlet is finished. The generated *exception handling clone of main* that executes the inlets, is shown in Figure 8.10. In reality, the Satin/Ibis compiler optimized the *instanceof* and *switch* statements away, as there is only one spawn location and one catch block in the example.

The second spawned *foo* will also throw an exception, which is handled as described above. However, *nrFinished* now reaches two, and an abort is issued in the inlet. Therefore, the Satin/Ibis runtime system will remove the last *foo* from the work queue, and all work is done. During the run, only one Java thread runs. The *main* method never runs concurrently with a *foo* method or an inlet. Therefore, no locks are needed on the critical path.

When running in parallel, exceptions thrown by a remote job are intercepted, and sent back to the CPU that spawned the method. The inlet is executed there, because that is where the localrecord of the spawner resides. On the receiving side, the incoming exceptions are delayed until a spawn or sync is reached, thus maintaining the invariant that there is only one thread running spawned work.

8.3.5   Aborts

We will now discuss the implementation of Satin/Ibis's abort mechanism on distributed-memory systems. The Satin/Ibis compiler recognizes calls to *ibis.satin.SatinObject.abort*, and replaces them with calls to the Satin/Ibis runtime system. The abort operation in the Satin/Ibis runtime system traverses the three lists described in section 8.3.2, and determines for each job in the lists whether it is dependent on the job that must be aborted (i.e., was spawned by it), using the parent stamps in the jobs. What action is taken depends on the list in which the job is found, but in all cases the spawn counter must be decreased by one, indicating that there is now one outstanding job less.

If a job that must be aborted is in the *onStack* list, the runtime system sets a bit in the invocation record of the job, signaling that the job is to be aborted. This bit is tested in the sync operation. If the bit is set, the generated code returns, and thus pops the work from the Java stack. When a job that is dependent on the job that must be aborted is found in the work queue, it can just be removed from the queue, as it was neither started yet nor stolen by a remote processor.

The difficult case is killing a job that was stolen (it is in the stolen list). In that case, it is removed from the stolen list, and an asynchronous abort message containing the stamp of the job to be killed is sent to the thief. Many race conditions are possible during this process. We avoid most of them by delaying abort messages at the receiving side, until the system reaches a safe point. When a spawn or a sync operation is reached, Satin/Ibis is in a safe state, and there is no other thread that runs work. At the safe points, the list of jobs to be aborted is traversed in the same way as described above. Again, all jobs are in one of the three lists.

Satin/Ibis guarantees that no results of outstanding spawned method invocations will be assigned after an abort operation, neither will any further inlets that are triggered by outstanding jobs be executed. Because the abort mechanism is implemented with asynchronous messages, special measures have to be taken to guarantee these strict semantics.

There are some race conditions that must be taken care of, but they can be handled in a straightforward way. For instance, an abort message for a job that has just finished may arrive. The Satin runtime system ignores the message, as the result has already been sent back to the owner of the work, and all state concerning the job has already been removed. Satin/Ibis knows this because it cannot find the state for the job (i.e., the invocation record) in one of the three lists described in Section 8.3.2, using the stamp of the job that must be aborted as identifier. Another race condition occurs when a result comes in for a job that has just been aborted. This can happen when an abort message and a job result message cross each other. Satin/Ibis can also detect this and ignore the result, as again the state has been removed, and the stamp of the job that must be aborted cannot be found in Satin/Ibis's tables.

There is a tradeoff between simplicity and efficiency here. In Satin/Ibis's implementation, local aborts are handled immediately, but remote aborts are delayed and only handled when a safe point is reached. The idea is that Satin/Ibis programs are fine grained, and sync operations are executed frequently. When a Satin program contains large pieces of sequential code, and the underlying system does not support interrupts, the programmer can optionally use a provided *poll* operation in a method that does not spawn work to poll the network for incoming messages and to force the runtime system to handle exceptions and aborts.

A nice property of our implementation is that all network messages that are sent to do an abort are *asynchronous*. This means that the machine that executed the abort can immediately continue working. This is especially advantageous in a wide-area setting with high latencies. The cost of an abort operation is virtually independent of network latency. However, the *benefit* of the abort operation does depend on the latency. During the latency, remote machined may be executing speculative work that has become unnecessary.

8.4   Performance Analysis

In this section, we will analyze the performance that Satin/Ibis achieves. We first present measurements on the DAS-2 system (see Section 1.9.2). Next, we show some results on a real grid: the European GridLab[1] testbed. On DAS-2, we use the IBM JIT, version 1.31 to run the Satin programs. We use an off-the-shelf JIT, to show that it is possible to achieve good performance with the current JVMs. We chose the IBM JIT, because it is the best performing JIT that is currently available on our platform. On the GridLab testbed, we used the JIT that was pre-installed on each particular system whenever possible, because this is what most users would probably do in practice. More details are given in Section 8.5.

We use the Ibis implementation on top of TCP for most measurements in this section. This means that the numbers shown below were measured using a 100% Java implementation. Therefore, they are interesting, because they give a clear indication of the performance level that can be achieved in Java with a "run everywhere" implementation, without using any native code. Whenever the Ibis implementation on Myrinet is used, it is explicitly stated.

8.4.1   Micro benchmarks

Table 8.2 shows the performance of the Satin/Ibis spawn and sync primitives, as well as the inlet mechanism. The numbers show that a single spawn operation, followed by a sync costs $0.18\,\mu$s on our hardware. Adding extra parameters costs only $0.03\,\mu$s extra, regardless of the type.

The cost of creating and throwing *Throwable* objects (e.g., exceptions) is important for Satin/Ibis, because the inlet mechanism uses exceptions. An interesting observation is that a new operation of a *Throwable* object (e.g., an exception) is extremely expensive (at least $2.98\,\mu$s). We found that this behavior is caused by the construction of the stack trace,

---

[1]See http://www.gridlab.org.

benchmark	time (μs)
normal spawn/sync, no parameter, no return	0.18
normal spawn/sync, 1 parameter + return value	0.21
new Throwable	2.98
new Throwable subclass without stack trace	0.15
new Throwable subclass without stack trace, with 1 field	0.15
call + exception return without stack trace	0.15
call + exception return without stack trace, with 1 field	0.15
inlet spawn/sync Throwable	5.22
inlet spawn/sync Throwable subclass with stack trace, with 1 field + return value	5.22
inlet spawn/sync cached Throwable	0.25
inlet spawn/sync cached Throwable subclass, with 1 field + return value	0.27
inlet spawn/sync new Throwable subclass without stack trace	0.40
inlet spawn/sync Throwable subclass without stack trace, with 1 field + return value	0.41

Table 8.2: Low-level benchmarks.

which is stored in the *Throwable* object. The cost of the *new* operation thus depends on the depth of the current execution stack. We can show this by overriding the *fillInStackTrace* method, which is responsible for the creation of the stack trace, with an empty method. This way, the cost of creating the stack trace is effectively eliminated. The cost of the *new* operation is reduced to only 0.15 μs, almost a factor of 20 better. Our test creates the *Throwable* objects directly in the *main* method. Therefore, when the depth of the execution stack is larger, this factor will be even larger. To circumvent the expensive stack trace creation (See Section 8.2) Satin/Ibis provides the *Inlet* class, which extends *Throwable* and overrides the *fillInStackTrace* method. When inlets are used, a stack trace is unused (it is useful only for debugging), and the programmer can use the *Inlet* class to avoid creating one.

A spawn operation that throws a newly created subclass of *Throwable* containing one field, followed by the execution of an empty inlet costs 5.22 μs, independent of the type of the parameter to the spawn and the result type stored in the *Throwable* object. Again, the cost is dominated by the creation of the stack trace (which is now deeper, because the exception is thrown in the spawned method). When a pre-allocated exception is used, the cost is reduced to only 0.25–0.27 μs, only about 0.06 μs more than a normal spawn operation. This shows the efficiency of the inlet design in Satin/Ibis, and of the code generated by the Satin/Ibis compiler. When a new exception is created, but now as a subclass of the *Inlet* class, the stack trace is not created and the cost is increased slightly to 0.4 μs, about two times slower than a normal spawn operation, but still 13 times faster than when a normal exception (with a stack trace) is used.

We can conclude from these numbers that Satin/Ibis's basic operations can indeed be implemented efficiently in pure Java, even on off-the-shelf JITs. In the following section we will investigate whether Satin/Ibis is also efficient at the application level.

application	problem size	# spawns	t_s (s)	t_1 (s)	avg. thread length	overhd factor
adaptive integration	0, 32E5, 1E-5	$1.4 \cdot 10^7$	5802.925	6003.109	0.416 ms	1.034
set covering problem	66, 33	$6.1 \cdot 10^4$	3218.644	3384.568	55.478 ms	1.052
fibonacci	47	$9.6 \cdot 10^9$	147.847	1582.840	0.165 μs	10.706
fibonacci threshold	52	$1.5 \cdot 10^5$	2060.501	2500.526	16.665 ms	1.214
iterative deepening A*	72	$1.5 \cdot 10^5$	1432.208	1442.829	9.671 ms	1.007
knapsack problem	32	$1.0 \cdot 10^6$	2945.411	3199.218	3.051 ms	1.086
matrix multiplication	4096 x 4096	$3.7 \cdot 10^4$	379.865	378.246	10.100 ms	0.996
n over k	39, 19	$3.2 \cdot 10^4$	1723.161	2069.319	63.154 ms	1.201
n-queens	22	$1.1 \cdot 10^5$	3271.527	3039.872	28.592 ms	0.929
prime factorization	19678904321	$4.2 \cdot 10^6$	2362.684	2310.028	0.551 ms	0.978
raytracer	balls2_hires.nff	$5.6 \cdot 10^6$	3535.431	3397.238	0.607 ms	0.961
traveling sales person	20	$1.1 \cdot 10^6$	5458.006	5010.499	4.457 ms	0.918

Table 8.3: Application overhead factors.

8.4.2   Applications

Table 8.3 shows problem sizes, number of spawn operations and the run times for the same applications that we used in Chapter 5 and 6. However, we enlarged the problem sizes to compensate for the faster hardware. We provide run times for the parallel version on a single machine and for the sequential version. We use this to calculate the overhead introduced by the parallel version. Furthermore, the table shows the average thread length. The sequential version is achieved by compiling with *javac* instead of *satinc* (this is correct because none of the programs in the table use inlets). In some cases, the parallel version is slightly faster than the sequential version, which we can only explain by caching effects and the internal behavior of the JIT, because the parallel version on a single machine does execute more code. The overhead for Fibonacci with Satin/Ibis is a factor of 10.7, which is slightly worse than the factor of 7.25 with Satin/Manta, but still considerably less than the factor of 61.5 which was reported for Atlas [14], which is also written in pure Java.

Figure 8.11 shows the parallel performance of the applications on Fast Ethernet, while Figure 8.12 shows the performance on Myrinet. In both cases, the Ibis serialization protocol is used. As explained in Section 5.3.2, we calculate the speedups relative to the sequential versions of the same applications. Nine of the twelve applications achieve a good speedup of more than 52 on 64 machines, even on Fast Ethernet. The speedup of IDA* is slightly worse, about 44 on 64 machines on Ethernet, and about 47 on Myrinet. This is caused by the relatively small problem size. With perfect speedup, IDA* would run only for 22 seconds on 64 machines. On Fast Ethernet, the run time of IDA* on 64 machines is 32.6 seconds, while on Myrinet, the run time is 31.1 seconds.

Matrix multiplication is solely limited by the network bandwidth. Therefore, the performance on Myrinet is much better than the performance on Fast Ethernet. In fact, on 64 machines, matrix multiplication is 2.9 times faster on Myrinet. The speedup of Fibonacci is almost perfect when we compare to the parallel version on a single machine, but the speedup relative to the sequential version is suboptimal due to the high sequential overhead factor (caused by the fine-grained spawning of work).
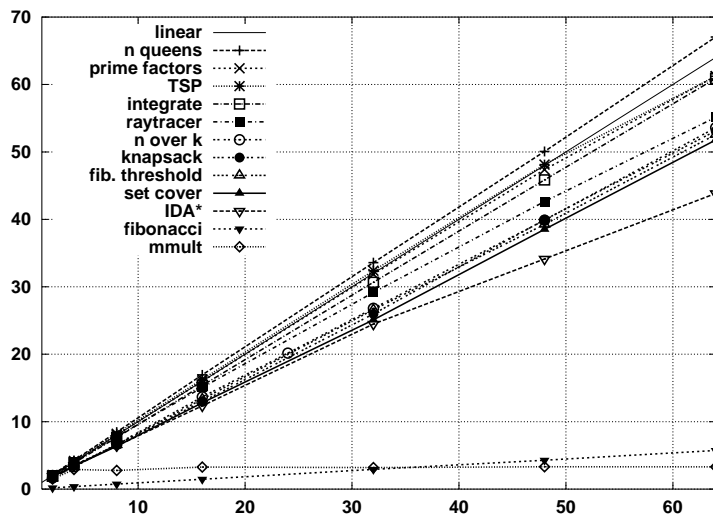
Figure 8.11: Application speedups with Ibis serialization on Fast Ethernet.

The graphs for Ibis serialization and standard Sun serialization (not shown) are virtually identical on Fast Ethernet, because both are able to use almost the full bandwidth of the Fast Ethernet network for arrays. For the applications that use more complicated data structures (e.g., TSP, IDA* and the raytracer), the performance with the Sun serialization protocol is slightly worse. On Myrinet, the performance with Sun serialization is about the same as the Fast Ethernet performance. This indicates that the bottleneck is in serialization, not in communication. The presented performance results for Fast Ethernet are especially interesting, because they show that good performance can be achieved with a 100% pure Java solution that runs on any JVM, and that can be easily deployed in a grid environment.

The Myrinet implementation becomes more important in a wide-area setting. As discussed in Chapter 6, CRS and ACRS achieve excellent performance on wide-area systems, but have higher communication requirements inside the local clusters. It is therefore advantageous to use the fast Myrinet network for local communication.

To verify the Satin/Ibis approach, we also ran the applications on four clusters of the wide-area DAS-2 system (see Section 1.9.2), using the real wide-area links. Because our implementation does not use any native code, we were able to run all applications without any modifications and, more importantly, even without having to recompile them for the different machines. We used the cluster-unaware RS algorithm (see Section 6.1.1) for load balancing. The speedups (relative to the sequential versions) on a single cluster of 64 machines and on four clusters of 16 machines are shown in Figure 8.13.

The results show that the performance on a single local system is virtually identical to the performance that is achieved on the wide-area DAS-2 system, even though we do not use one of Satin's cluster-aware load-balancing algorithms. These good results are in
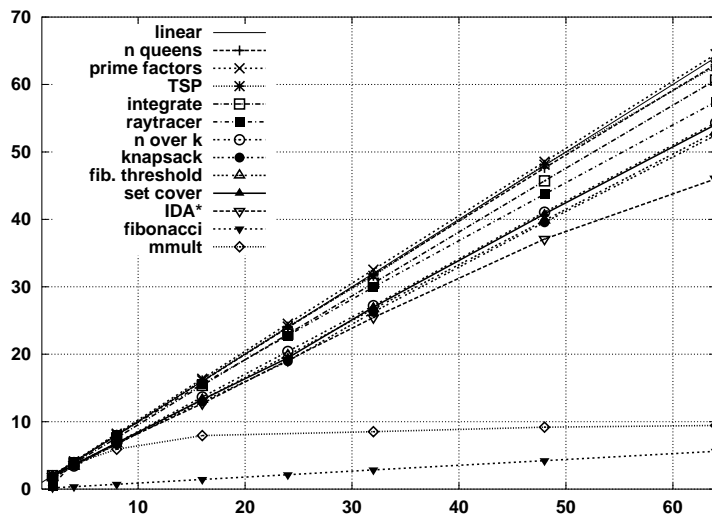
Figure 8.12: Application speedups with Ibis serialization on Myrinet.

source / destination	to VU Amsterdam	to Nikhef Amsterdam	to Leiden	to Delft
latency from				
VU Amsterdam	—	1.3	2.7	2.4
Nikhef Amsterdam	1.1	—	1.7.	1.5
Leiden	2.7	1.7	—	3.7
Delft	2.5	1.5	3.8	—
throughput from				
VU Amsterdam	—	11.1	38.1	39.9
Nikhef Amsterdam	11.2	—	10.6	11.2
Leiden	37.8	11.1	—	24.1
Delft	35.9	11.2	24.2	—

Table 8.4: Round-trip wide-area latencies between the DAS-2 clusters in milliseconds, throughputs in MByte/s.

fact not surprising: the DAS-2 system is hierarchical, but is hardly a wide-area system, as all clusters are located within the Netherlands. Table 8.4 shows the round-trip wide-area latencies and throughputs of the DAS-2 system. The latencies are measured with *ping*, while the bandwidths are measured by *netperf* [2], using 32 KByte packets. The numbers are provided in both directions, because outbound and incoming channels can be routed differently. All wide-area round-trip latencies of the DAS-2 system are between 1 and 4 milliseconds, while in Chapter 6, we used (simulated) wide-area latencies of 10 and even 100 milliseconds. Also, the bandwidth between the DAS-2 clusters is relatively high: between 10.6 and 39.9 MByte/s, while we used WAN links of 0.1 - 1 MByte/s
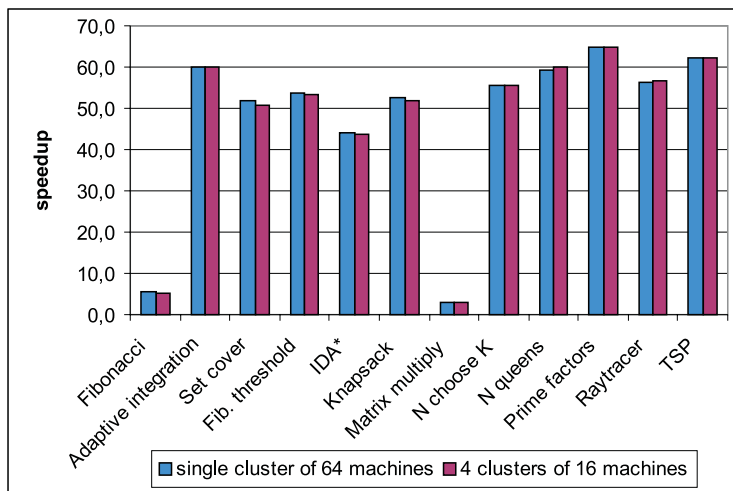
---
[2]See http://www.netperf.org.

Figure 8.13: Application speedups with Ibis serialization on the wide-area DAS-2 system.

in our simulations in Chapter 6. Figures 6.5 and 6.6 in Chapter 6 indeed show that RS still performs reasonably with (simulated) wide-area latencies of only 10 milliseconds and a WAN bandwidth of 1 MByte/s. Therefore, the results presented here confirm the simulation results shown in Chapter 6.

8.5 Experiences with Satin on a Real Grid Testbed

In this section, we will present a case study to analyze the performance that Satin/Ibis achieves in a real grid environment. We ran the raytracer application on the European GridLab [2] testbed. More precisely, we were using a characteristic subset of the machines on this testbed that was available for our measurements at the time the study was performed. Because simultaneously starting and running a parallel application on multiple clusters still is a tedious and time-consuming task, we had to restrict ourselves to a single test application. We have chosen the raytracer for our tests as it is sending the most data of all our applications, making it very sensitive to network issues. The picture that is generated by the raytracer application is shown in Figure 2.11. To achieve a more realistic grid scenario, we use a higher resolution (4096 × 4096, with 24-bit color) than in the previous chapters. It takes approximately 10 minutes to solve this problem on our testbed. The resulting image was used for the cover of this thesis.

This is an interesting experiment for several reasons. Firstly, the testbed contains machines with several different architectures; Intel, SPARC, MIPS, and Alpha processors are used. Some machines are 32 bit, while others are 64 bit. Also, different operating systems and JVMs are in use. Therefore, this experiment is a good method to investigate whether Java's "run everywhere" feature really works in practice. The assumption that

location	architecture	Operating System	JIT	nodes	CPUs / node	total CPUs
Vrije Universiteit *Amsterdam* *The Netherlands*	Intel Pentium-III 1 GHz	Red Hat Linux kernel 2.4.18	IBM 1.4.0	8	1	8
Vrije Universiteit *Amsterdam* *The Netherlands*	Sun Fire 280R UltraSPARC-III 750 MHz 64 bit	Sun Solaris 8	SUN HotSpot 1.4.2	1	2	2
ISUFI/High Perf. Computing Center *Lecce, Italy*	Compaq Alpha 667 MHz 64 bit	Compaq Tru64 UNIX V5.1A	HP 1.4.0 based on HotSpot	1	4	4
Cardiff University *Cardiff, Wales, UK*	Intel Pentium-III 1 GHz	Red Hat Linux 7.1 kernel 2.4.2	SUN HotSpot 1.4.1	1	2	2
Masaryk University, *Brno, Czech Republic*	Intel Xeon 2.4 GHz	Debian Linux kernel 2.4.20	IBM 1.4.0	4	2	8
Konrad-Zuse-Zentrum für Informationstechnik *Berlin, Germany*	SGI Origin 3000 MIPS R14000 500 MHz	IRIX 6.5	SGI 1.4.1-EA based on HotSpot	1	16	16

Table 8.5: The GridLab testbed.

this feature successfully hides the complexity of the different underlying architectures and operating systems, was the most important reason for investigating the Java-centric solutions presented in this thesis. It is thus important to verify the validity of this claim.

Secondly, the machines are connected by the Internet. The links show typical wide-area behavior, as the physical distance between the sites is large. For instance, the distance from Amsterdam to Lecce is roughly 2000 kilometers (about 1250 miles). Figure 8.14 shows a map of Europe, annotated with the machine locations. This gives an idea of the distances between the sites. We use this experiment to verify Satin's load-balancing algorithms in practice, with *real* non-dedicated wide-area links. We have run the raytracer both with the standard random stealing algorithm (RS) and with the new cluster-aware algorithm (CRS) that was introduced in Chapter 6. For practical reasons, we had to use relatively small clusters for the measurements in this section. The simulation results in Section 6.4 show that the performance of CRS increases when larger clusters are used, because there is more opportunity to balance the load inside a cluster during wide-area communication.

Some information about the machines we used is shown in Table 8.5. Because the sites are connected via the Internet, we have no influence on the amount of traffic that flows over the links. To reduce the influence of Internet traffic on the measurements, we performed measurements after midnight (CET). However, in practice there still is some variability in the link speeds. We measured the latency of the wide-area links by running *ping* 50 times, while the bandwidth is measured with *netperf* [3], using 32 KByte packets. The measured latencies and bandwidths are shown in Table 8.6. All sites had difficulties from time to time while sending traffic to Lecce, Italy. For instance, from Amsterdam to Lecce, we measured latencies from 44 milliseconds up to 3.5 seconds.
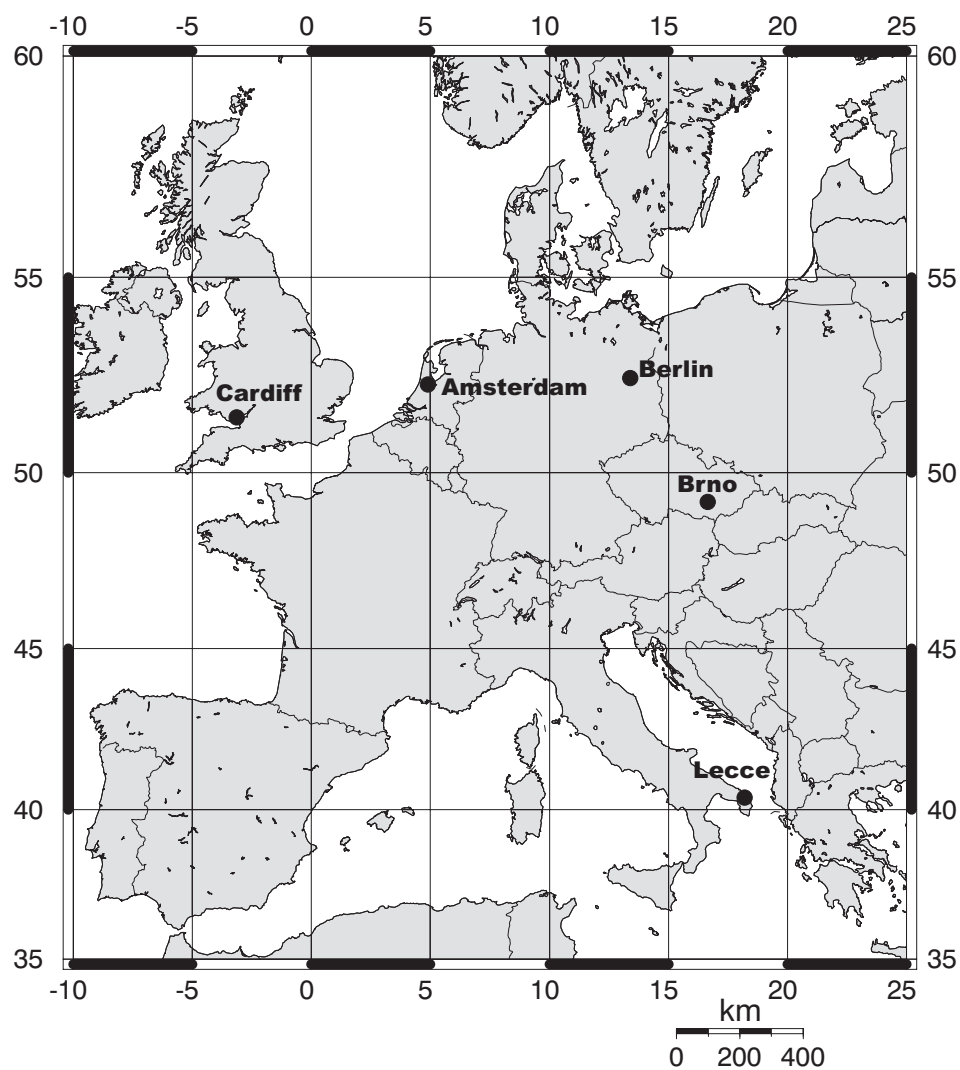
---
[3]See http://www.netperf.org.

Figure 8.14: Locations of the GridLab testbed sites used for the experiments.

destination source	to A'dam DAS-2	to A'dam Sun	to Lecce	to Cardiff	to Brno	to Berlin
daytime latency from						
A'dam DAS-2	—	1	204	16	20	42
A'dam Sun	1	—	204	15	19	43
Lecce	198	195	—	210	204	178
Cardiff	9	9	198	—	28	26
Brno	20	20	188	33	—	22
Berlin	18	17	185	31	22	—
daytime bandwidth from						
A'dam DAS-2	—	11338	42	750	3923	2578
A'dam Sun	11511	—	22	696	2745	2611
Lecce	73	425	—	44	43	75
Cardiff	842	791	29	—	767	825
Brno	3186	2709	26	588	—	2023
Berlin	2555	2633	9	533	2097	—
nighttime latency from						
A'dam DAS-2	—	1	65	15	20	18
A'dam Sun	1	—	62	14	19	17
Lecce	63	66	—	60	66	64
Cardiff	9	9	51	—	27	21
Brno	20	19	64	33	—	22
Berlin	18	17	59	30	22	—
nighttime bandwidth from						
A'dam DAS-2	—	11442	40	747	4115	2578
A'dam Sun	11548	—	46	701	3040	2626
Lecce	77	803	—	94	110	82
Cardiff	861	818	37	—	817	851
Brno	3167	2705	37	612	—	2025
Berlin	2611	2659	9	562	2111	—

Table 8.6: Round-trip wide-area latencies (in milliseconds) and bandwidths (in KByte/s) between the GridLab sites.

Also, we experienced packet loss with this link: up to 23% of the packets was dropped along the way. We also performed the same measurement during daytime, to investigate how regular Internet traffic influences the application performance. The measurements show that there can be more than a factor of two difference in link speeds during daytime and nighttime, especially the links from and to Lecce show a large variability. It is also interesting to see that the link performance from Lecce to the two sites in Amsterdam is different. We verified this with *traceroute*, and found that the traffic is indeed routed differently. It is likely that this is caused by the fact that the two machines in Amsterdam use an IP-address in a different range. The DAS-2 file server uses 130.37.199.2, while the address of the SUN server is 192.31.231.65.

Ibis, Satin and the raytracer application were all compiled with the standard Java compiler *javac* on the DAS-2 machine in Amsterdam, and then just copied to the other GridLab sites, without recompiling or reconfiguring anything. On most sites, this works flawlessly. However, we did run into several practical problems. A summary is given in Table 8.7. Some of the GridLab sites have firewalls installed, which block Satin's traffic when no special measures are taken. Most sites in our testbed have some open port range,

problem	solution
Firewalls block Ibis communication.	Bind all sockets to ports in the open range.
Buggy JITs.	Upgrade to Java 1.4 JITs.
Machines with multiple IP addresses.	Use a single, externally valid IP address.

Table 8.7: Problems encountered in a real grid environment, and their solutions.

site	architecture	run time (s)	relative node speed	relative total speed of cluster	% of total system
A'dam DAS-2	1 GHz Intel Pentium-III	233.1	1.000	8.000	32.4
A'dam Sun	750 MHz UltraSPARC-III	445.2	0.523	1.046	4.2
Lecce	667 MHZ Compaq Alpha	512.7	0.454	1.816	7.4
Cardiff	1 GHz Intel Pentium-III	758.9	0.307	0.614	2.5
Brno	2.4 GHz Intel Xeon	152.8	1.525	12.200	49.5
Berlin	500 MHz MIPS R14000	3701.4	0.062	0.992	4.0
total				_24.668_	100.0

Table 8.8: Relative speeds of the machine and JVM combinations in the testbed.

which means that traffic to ports within this range can pass through. The solution we use to avoid being blocked by firewalls is straightforward: all sockets used for communication in Ibis are bound to a port within the (site-specific) open port range. We are working on a more general solution that multiplexes all traffic over a single port. Another solution is to multiplex all traffic over a (Globus) ssh connection, as is done by Kaneda et al. [85], or using a mechanism like SOCKS [104].

Another problem we encountered was that the JITs installed on some sites contained bugs. Especially the combination of threads and sockets presented some difficulties. There seems to be a bug in Sun's 1.3 JIT (HotSpot) related to threads and socket communication. In some circumstances, a blocking operation on a socket would block the whole application instead of just the thread that does the operation. The solution for this problem was to upgrade to a Java 1.4 JIT, where the problem is solved.

Finally, some machines in the testbed are multi-homed: they have multiple IP addresses. The original Ibis implementation on TCP got confused by this, because the _InetAddress.getLocalHost_ method can return an IP address in a private range, or an address for an interface that is not accessible from the outside. Our current solution is to manually specify which IP address has to be used when multiple choices are available. All machines in the testbed have a Globus [56] installation, so we used GSI-SSH (Globus Security Infrastructure Secure Shell) [58] to login to the GridLab sites. We had to start the application by hand, as not all sites have a job manager installed. When a job manager is present, Globus can be used to start the application.

As shown in Table 8.5, we used 40 processors in total, using 6 machines located at 5 sites all over Europe, with 4 different processor architectures. After solving the aforementioned practical problems, Satin on the TCP Ibis implementation ran on all sites, in pure Java, without having to recompile anything.

As a benchmark, we first ran the parallel version of the raytracer with a smaller problem size (512×512, with 24 bit color) on a single machine on all individual clusters.

This way, we can compute the relative speeds of the different machines and JVMs. The results are presented in Table 8.8. To calculate the relative speed of each machine/JVM combination, we normalized the run times relative to the run time of the raytracer on a node of the DAS-2 cluster located in Amsterdam. It is interesting to note that the quality of the JIT compiler can have a large impact on the performance at the application level. A node in the DAS-2 cluster and the machine in Cardiff are both 1 GHz Intel Pentium-IIIs, but there is more than a factor of three difference in application performance. This is as least partly caused by the different JIT compilers that were used. On the DAS-2, we used the more efficient IBM 1.4 JIT, while the SUN 1.4 JIT (HotSpot) was installed on the machine in Cardiff.

Furthermore, the results show that, although the clock frequency of the machine at Brno is 2.4 times as high as the frequency of a DAS-2 node, the speed improvement is only 53%. Both machines use Intel processors, but the Xeon machine in Brno is based on Pentium-4 processors, which do less work per cycle than the Pentium-III CPUs that are used by the DAS-2. It is in general not possible to use the clock frequencies to compare processor speeds.

Finally, it is clear that the Origin machine in Berlin is slow compared to the other machines. This is partly caused by the inefficient JIT, which is based on the SUN HotSpot JVM. Because of the combination of slow processors and the inefficient JIT, the 16 nodes of the Origin we used are about as fast as a single 1 GHz Pentium-III with the IBM JIT. The Origin machine thus hardly contributes anything to the computation. The table shows that, although we used 40 CPUs in total for the grid run, the relative speed of these processors together adds up to 24.668 DAS-2 nodes (1 GHz Pentium-IIIs). The percentage of the total compute power that each individual cluster delivers is also shown in Table 8.8.

We also ran the raytracer on a single DAS-2 machine, with the large problem size that we will use for the grid runs. This took 13746 seconds (almost four hours). The sequential program without the Satin constructs takes 13564 seconds, the overhead of the parallel version thus is about 1%. With perfect speedup, the run time of the parallel program on the GridLab testbed would be 13564 divided by 24.668, which is 549.8 seconds (about nine minutes). We call this upper bound on the performance that can be achieved on the testbed $t_{perfect}$. We can use this number to calculate the efficiency that is achieved by the real parallel runs. We call the actual run time of the application on the testbed t_{grid}. Efficiency can be defined as follows:

$$efficiency = \frac{t_{perfect}}{t_{grid}} * 100\%$$

We have also measured the time that is spent in communication (t_{comm}). This includes idle time, because all idle time in the system is caused by waiting for communication to finish. We calculate the relative communication overhead with this formula:

$$communication\ overhead = \frac{t_{comm}}{t_{perfect}} * 100\%$$

algorithm	run time (s)	communication		parallelization		efficiency
		time (s)	overhead	time (s)	overhead	
nighttime						
RS	877.6	198.5	36.1%	121.9	23.5%	62.6%
CRS	676.5	35.4	6.4%	83.9	16.6%	81.3%
daytime						
RS	2083.5	1414.5	257.3%	111.8	21.7%	26.4%
CRS	693.0	40.1	7.3%	95.7	18.8%	79.3%
single cluster 25						
RS	579.6	11.3	2.0%	11.0	1.9%	96.1%

Table 8.9: Performance of the raytracer application on the GridLab testbed.

alg.	intra cluster		inter cluster	
	messages	MByte	messages	MByte
nighttime				
RS	3218	41.8	11473	137.3
CRS	1353295	131.7	12153	86.0
daytime				
RS	56686	18.9	149634	154.1
CRS	2148348	130.7	10115	82.1
single cluster 25				
RS	45458	155.6	—	—

Table 8.10: Communication statistics for the raytracer application on the GridLab testbed.

Finally, the time that is lost due to parallelization overhead (t_{par}) is calculated as shown below:

$$t_{par} = t_{grid} - t_{comm} - t_{perfect}$$

$$parallelization\ overhead = \frac{t_{par}}{t_{perfect}} * 100\%$$

The results of the grid runs are shown in Table 8.9. For reference, we also provide measurements on a single cluster, using 25 nodes of the DAS-2 system. The results presented here are the fastest runs out of three experiments. During daytime, the performance of the raytracer with RS showed a large variability, some runs took longer than an hour to complete, while the fastest run took about half an hour. Therefore, in this particular case, we took the best result of six runs. This approach thus is in favor of RS. With CRS, this effect does not occur: the difference between the fastest and the slowest run during daytime was less than 20 seconds. During night, when there is little Internet traffic, the application with CRS is already more than 200 seconds faster (about 23%) than with the RS algorithm. During daytime, when the Internet links are heavily used, CRS outperforms RS by a factor of three. Regardless of the time of the day, the efficiency of a parallel run with CRS is about 80%.

The numbers in Table 8.9 show that the parallelization overhead on the testbed is significantly higher compared to a single cluster. Sources of this overhead are thread creation

and switching caused by incoming steal requests, and the locking of the work queues. The overhead is higher on the testbed, because five of the six machines we use are SMPs (i.e. they have a shared memory architecture). In general, this means that the CPUs in such a system have to share resources, making memory access and especially synchronization potentially more expensive. The latter has a negative effect on the performance of the work queues. Also, multiple CPUs share a single network interface, making access to the communication device more expensive. The current implementation of Satin treats SMPs as clusters (i.e., on a N-way SMP, we start N JVMs). Therefore, Satin pays the price of the SMP overhead, but does not exploit the benefits of SMP systems, such as the available shared memory. An implementation that does utilize shared memory when available is planned for the future.

Communication statistics of the grid runs are shown in Table 8.10. The numbers in the table totals for the whole run, summed over all CPUs. Again, statistics for a single cluster run are included for reference. The numbers show that almost all of the overhead of RS is in excessive wide-area communication. During daytime, for instance, it tries to send 154 MByte over the busy Internet links. During the time-consuming wide-area transfers, the sending machine is idle, because the algorithm is synchronous. CRS sends only about 82 MBytes over the wide-area links (about half the amount of RS), but more importantly, the transfers are asynchronous. With CRS, the machine that initiates the wide-area traffic concurrently tries to steal work in the local cluster, and also concurrently executes the work that is found.

CRS effectively trades less wide-area traffic for more local communication. As shown in Table 8.10, the run during the night sends about 1.4 million local-area messages. During daytime, the CRS algorithm has to do more effort to keep the load balanced: during the wide-area steals, about 2.1 million local messages are sent while trying to find work within the local clusters. This is about 60% more than during the night. Still, only 40.1 seconds are spent communicating. With CRS, the run during daytime only takes 16.5 seconds (about 2.4%) longer than the run at night. The total communication overhead of CRS is at most 7.3%, while with RS, this can be as much as two thirds of the run time (i.e. the algorithm spends more time on communicating than on calculating useful work).

Because all idle time is caused by communication, the time that is spent on the actual computation can be calculated by subtracting the communication time from the actual run time (t_{grid}). Because we have gathered the communication statistics per machine (not shown), we can calculate the total time a whole *cluster* spends computing the actual problem. Given the amount of time a cluster performs useful work and the relative speed of the cluster, we can calculate what fraction of the total work is calculated by each individual cluster. We can compare this workload distribution with the ideal distribution which is represented by the rightmost column of Table 8.8. The ideal distribution and the results for the four grid runs are shown in Figure 8.15. The difference between the perfect distribution and the actual distributions of the four grid runs is hardly visible. From the figure, we can conclude that, although the workload distribution of both RS and CRS is virtually perfect, the RS algorithm itself spends a large amount of time on *achieving* this distribution. CRS does not suffer from this problem, because wide-area traffic is asynchronous and is overlapped with useful work that was found locally. Still, it achieves an almost optimal distribution.
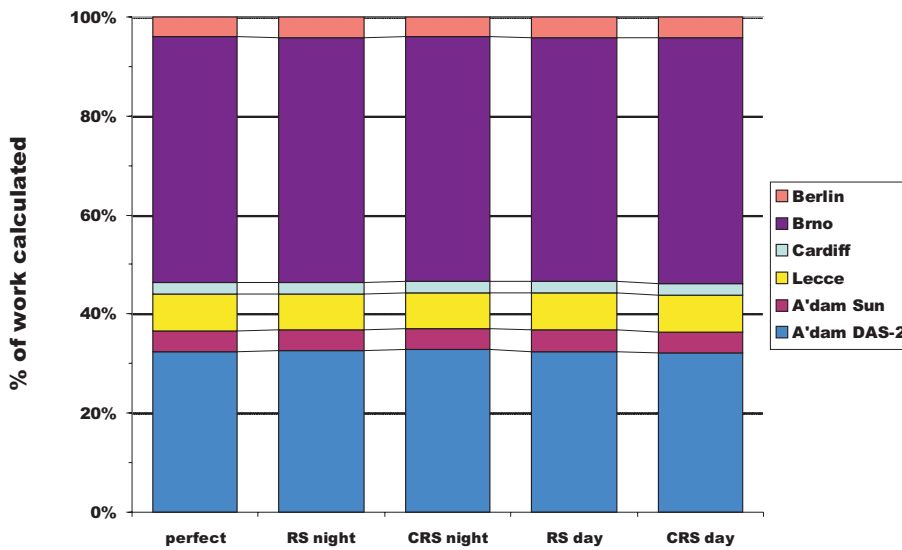
Figure 8.15: Distribution of work over the different sites.

To summarize, the experiment described in this section shows that the Java-centric approach to grid computing, and the Satin/Ibis system in particular, works extremely well in practice in a real grid environment. It took hardly any effort to run Ibis and Satin on a heterogeneous system. Furthermore, the performance results clearly show that CRS outperforms RS in a real grid environment, especially when the wide-area links are also used for other (Internet) traffic. With CRS, the system is idle (waiting for communication) during only a small fraction of the total run time. We expect even better performance when larger clusters are used (See Section 6.4).

8.6 Satin and Speculative Parallelism: a Case Study

To test Satin/Ibis's new abort mechanism, we implemented sequential and parallel versions of the Awari game [43] (see also Section 2.7.18), using the MTD(f) [133] algorithm. This application contains a large amount of extremely speculative parallelism. A transposition table [153] is used to avoid evaluating identical game positions multiple times. The sequential code for the search algorithm used to implement Awari is shown in Figure 8.16.

The algorithm works a follows. First, on lines 2–5, the stop condition is tested. The search stops when the depth reaches zero. When the stop condition evaluates to false, the children of the current node are generated. Next, a transposition-table lookup is done (line 10) to check whether a solution has already been found for this game position. If this is

```
1  NodeType depthFirstSearch(NodeType node, int pivot, int depth) {
2      if(depth == 0) { // Stop if the depth is 0.
3          node.evaluate();
4          return null;
5      }
6      NodeType[] children = node.generateChildren();
7
8      // Check transposition table for a solution or a promising child.
9      short bestChild = 0, promisingChild = 0;
10     TranspositionTableEntry e = tt.lookup(node.signature);
11     if(e != null && node.signature == e.tag) {
12         if(e.depth >= depth) {
13             if((e.lowerBound ? e.value >= pivot : e.value < pivot)) {
14                 node.score = e.value;
15                 return children[e.bestChild];
16             }
17         }
18         bestChild = promisingChild = e.bestChild;
19     }
20
21     // Try promising child first, it may generate a cut-off.
22     depthFirstSearch(children[promisingChild], 1-pivot, depth-1);
23     node.score = -children[promisingChild].score;
24     if(node.score >= pivot) {
25         tt.store(node, promisingChild, depth);
26         return children[promisingChild];
27     }
28
29     // Search remaining children.
30     for(short i = 0; i < children.length; i++) {
31         if(i == promisingChild) continue;
32         depthFirstSearch(children[i], 1-pivot, depth-1);
33         if(-children[i].score > node.score) {
34             bestChild = i;
35             node.score = -children[i].score;
36             if(node.score >= pivot) break;
37         }
38     }
39     tt.store(node, bestChild, depth);
40     return children[bestChild];
41 }
```

Figure 8.16: Sequential Awari pseudo code.

the case, the current node can be pruned (lines 11–17). When this position was evaluated before, but using a smaller depth, we use the transposition table to retrieve the node that gave the best result for the smaller depth (line 18). It is likely that this node will also be the best with the current (larger) depth.

This child that promises to be a good solution (or the first child when no transposition-table entry was found) is evaluated first, because it may generate a cutoff. This way, we can avoid searching the other children when the node does indeed lead to a good

```
1  public void spawn_depthFirstSearch(NodeType node, int pivot,
2                        int depth, short currChild) throws Done {
3      depthFirstSearch(node, pivot, depth);
4      throw new Done(node.score, currChild);
5  }
6
7  NodeType depthFirstSearch(NodeType node, int pivot, int depth) {
8      // The first part of the sequential algorithm is unchanged...
9
10     // Search remaining children.
11     for(short i = 0; i<children.length; i++) {
12         if(i == promisingChild) continue;
13         try {
14             spawn_depthFirstSearch(children[i], 1-pivot, depth-1, i);
15         } catch (Done d) {                              // The inlet.
16             if(-d.score > node.score) {
17                 bestChild = d.currChild;
18                 node.score = -d.score;
19                 if(node.score >= pivot) abort();
20             }
21             return null; // Exit the inlet, do not fall through.
22         }
23     }
24     sync();
25     tt.store(node, bestChild, depth);
26     return children[bestChild];
```

Figure 8.17: Parallel Awari pseudo code.

position (lines 21–27). Finally, the remaining children are searched in the order they were generated (lines 29–38). Again, when a good solution is found, the search can be stopped immediately, without looking at the remaining children. This is implemented with the *break* statement in line 36. In all cases, the best result is stored in the transposition table (on lines 25 and 39) for future reference.

It is relatively straightforward to parallelize the search algorithm in Satin/Ibis. The parallel version of the MTD(f) code is shown in Figure 8.17, while some additional code that is needed is shown in Figure 8.18. An extra method called *spawn_depthFirstSearch* is shown in Figure 8.17. The method contains code to call *depthFirstSearch*, and to return the result with an exception instead of a normal return value to trigger an inlet. In the interface *MtdfInterface* (shown in Figure 8.18), *spawn_depthFirstSearch* is marked as a spawnable method.

The code for the parallel search algorithm is largely identical to the sequential code. Only the last part of the algorithm (lines 30–38 in Figure 8.16) that evaluates the remaining children when no cutoff is generated has been changed, and only this part is shown in Figure 8.17. The first child is examined sequentially, because it may generate a cutoff. The remaining children are searched in parallel, because the recursive method invocation on line 14 is now a spawned method invocation. Thus, the parallel version speculatively spawns all work, and then waits for the results to arrive at the sync statement (line 24).

```
1  final class Done extends Inlet implements java.io.Serializable {
2      short score;
3      short currChild;
4
5      Done(short score, short currChild) {
6          this.score = score;
7          this.currChild = currChild;
8      }
9  }
10
11 public interface MtdfInterface extends ibis.satin.Spawnable {
12     public void spawn_depthFirstSearch(NodeType node, int pivot,
13                 int depth, short currChild) throws Done;
14 }
```
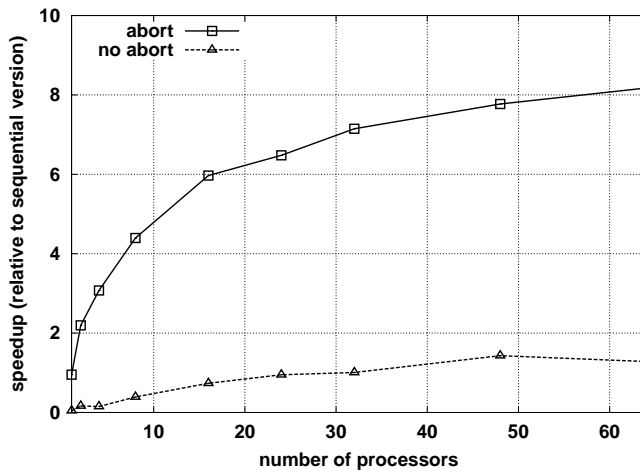
Figure 8.18: Additional code needed for parallel Awari.

Figure 8.19: Speedups of Awari, with and without the abort mechanism.

Note that, because the first part of the algorithm is unchanged, the cost of spawning work is paid only when no cutoff occurs.

When a spawned task is finished, it throws an exception of type *Done* (shown in Figure 8.18), which contains the result. The exception triggers the *inlet* on lines 15–22 (Figure 8.17). The inlet contains the same if-statement as the original sequential version, but reads the result from the exception, and executes an *abort* instead of a *break* when a good solution has been found. The effect is the same: the other children of the current node will not be evaluated anymore.

Essentially, parallelism is introduced in Awari by speculatively searching multiple game trees in parallel. To achieve good performance, search overhead (the result of spec-

ulatively searching subtrees) should be minimized. If some processor finds a promising solution, it should be forwarded to the others, allowing them to prune work. Therefore, the transposition table is replicated. While it is not possible to express this in Satin/Ibis itself, it is possible to combine Satin/Ibis's primitives with any other communication mechanism, such as RepMI or GMI. Unfortunately, an Ibis implementation of the latter programming models is not yet available. Therefore, for this case study, we used TCP/IP sockets to implement a simple replicated transposition table. To reduce communication overhead, it is possible to only replicate the results up to a certain depth from the root. Message combining is used to aggregate multiple transposition-table updates into a singe network message, to avoid excessive communication. Transposition-table updates are broadcast asynchronously. We used a transposition table with 16.8 million entries, occupying about 288 MBytes memory. The Awari program used in this chapter is not a state-of-the-art implementation, but we use it only to show that Satin/Ibis's abort mechanism is useful.

8.6.1 Performance Evaluation

The problem we used starts from the initial position, and uses a search depth of 21. Our sequential implementation (and the Satin/Ibis version on a single machine) visits 850,868,742 different positions. The parallel runs can visit more nodes, because of the speculative nature of the search. The parallel version can in theory also search fewer nodes, since the search order is not deterministic anymore. A speculatively spawned job can produce a good result quickly, causing more nodes to be pruned. The Satin/Ibis version on a single machine is only 4.9% slower than the sequential version, even though some variables in the *depthFirstSearch* method have to be accessed via a localrecord, because they are potentially used inside the inlet. Our sequential implementation visits about 353,000 states per second. The Satin/Ibis version on a single machine is only slightly slower, and still visits 337,000 states per second.

Figure 8.19 shows the speedups of Awari both using Satin/Ibis's abort mechanism, and without aborts, relative to the sequential implementation. The speedup with aborts on 64 machines is about 8.2. This may seem disappointing, but game-tree search in general is hard to parallelize, and Awari is even harder, because of its small branching factor. Other researchers report similar speedups [90]. When no abort mechanism is used, Awari hardly achieves any speedup at all.

Figures 8.20 and 8.21 show a detailed breakdown of the parallel runs. The graphs show the normalized aggregated execution time relative to the sequential version. This way, the absolute overhead factors can be quantified. Thus, with perfect speedups, the height of all bars would be 1. For instance, the graph shows that the parallel run with aborts on 64 machines has a total overhead factor of about 7.8, hence the speedup of 64 / 7.8 = 8.2. When Satin/Ibis's abort mechanism is used, the search overhead on 64 machines is about a factor of 4. However, without aborts, the search overhead is a factor of 15.8, almost four times higher.

The idle times are caused by load imbalance, and increase with the number of machines, both with and without aborts. This is partly caused by the communication overhead of replicating the transposition table, which uses the same communication channel
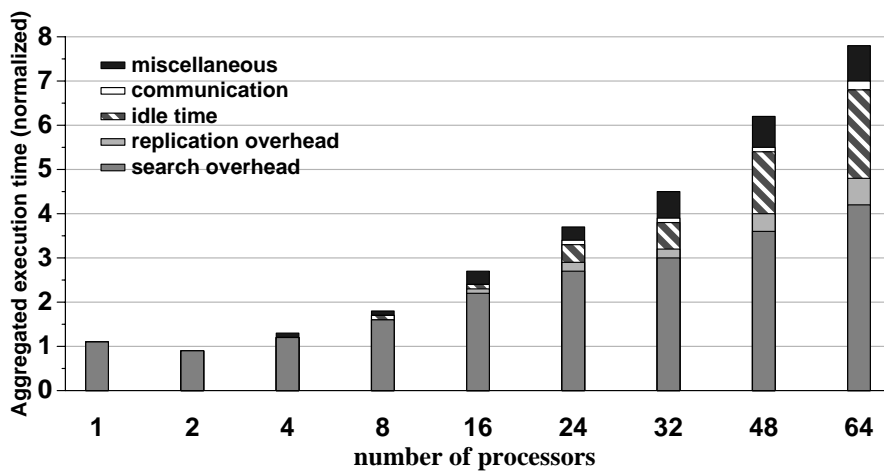
Figure 8.20: Breakdown of Awari with Satin/Ibis's abort mechanism.

as Satin/Ibis's work stealing algorithm. Also, due to the small branching factor of Awari, there sometimes is not enough parallelism to keep all machines busy.

The "miscellaneous" overhead in Figures 8.20 and 8.21 is partly caused by the lookups and stores in the transposition table. For instance, the runs on 2 and 4 machines without aborts do an order of magnitude more lookups than the version with the abort mechanism enabled. The high overhead of transposition-table lookups and stores shows that the abort mechanism is a more efficient way of reducing search overhead.

On 64 machines, about 640 thousand jobs were aborted, and about 83 thousand abort messages were sent over the network. Still, the Satin/Ibis runtime system spent only 220 milliseconds per machine aborting work. The results show that Satin/Ibis's abort mechanism can be implemented efficiently, even on distributed memory systems. Furthermore, the case study shows that the mechanism is useful, because it allows the programmer to easily express speculative parallelism.

8.7   Related Work

We discussed Satin/Ibis, a divide-and-conquer extension of Java that provides (amongst others) asynchronous exceptions and an abort mechanism. Satin/Ibis is designed for wide-area systems, without shared memory. Satin/Ibis is implemented in 100% pure Java, so the only requirement to run Satin/Ibis is a JVM. This facilitates the deployment of Satin/Ibis on the grid.
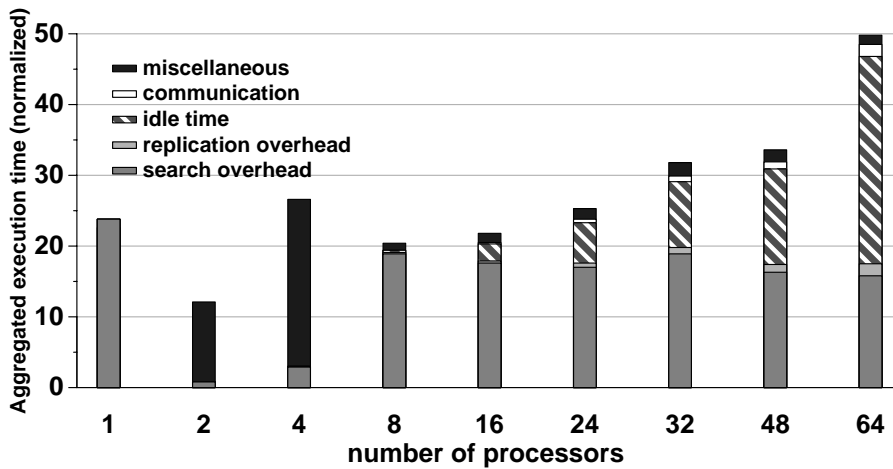
Figure 8.21: Breakdown of Awari without an abort mechanism.

Standard Java provides a method called *java.lang.Thread.interrupt* to asynchronously signal a *waiting* thread (e.g., a thread that is blocked on an I/O operation). When a thread is not waiting, only a flag is set, and no exception is thrown. Thus, this mechanism cannot be used to implement an abort mechanism. Older Java versions also specified methods to stop, suspend and resume threads. However, these were later deprecated, because they were inherently unsafe [68]. Satin/Ibis avoids this, because inlets do not run concurrently with spawned work, therefore avoiding the use of a locking mechanism.

The Java classes by Lea et al. [100] can also be used for divide-and-conquer algorithms. However, they are restricted to shared-memory systems, and an abort mechanism is not provided.

Several divide-and-conquer systems exist that are based on the C language. Among them, Cilk [22] supports inlets and aborts, but only for shared-memory machines. Cilk-NOW or distributed Cilk [24] and DCPAR [61] run on local-area, distributed-memory systems, but neither support shared data, inlets nor aborts. SilkRoad [129] is a version of Cilk for distributed memory that uses a software DSM to provide shared memory to the programmer, targeting at small-scale, local-area systems. As SilkRoad is based on distributed Cilk, an abort primitive is not provided.

A modification to make the Cilk scheduler more efficient on a system that contains machines with different speeds is discussed in [15]. The improved algorithm lets faster machines steal work that is currently being worked on by slower machines. By aborting the work and restarting it on the faster machine, the problem that the system is idle except for a single large job running on a slow machine is circumvented. Our experience on a real

grid system containing machines with a factor of 20 difference in speed suggests that this problem does not necessarily occur in practice. In fact, the work distribution with RS and CRS was almost perfect. It is possible, however, that more course grained applications do suffer from this problem. Unfortunately, performance results are not given in [15].

Marlow et al. [115] extend the Haskell language with asynchronous exceptions. However, their mechanism is targeted at error handling (e.g., timeouts, etc). The implementation of the mechanism is complicated because exceptions can be received when locks are taken. Moreover, safe regions (where no exceptions can be received) are introduced into the language. Posix threads [32] also support asynchronous signals. However, it suffers from the same problems as the asynchronous exceptions in Haskell.

Satin/Ibis avoids the aforementioned issues: inlets do not run concurrently with other work, and the programmer does not have to annotate the code to mark safe regions. This is possible because Satin/Ibis conceptually creates a *new* thread to handle the exception, whereas Haskell and Posix threads interrupt a running thread. The new thread that is started by Satin/Ibis when an asynchronous exception occurs shares its local variables and parameters with the spawner. Therefore, the new thread can access the state of the thread that received the exception, making our approach as expressive as the traditional solution of interrupting a thread.

8.8   Conclusions

In this chapter, we have applied the lessons we learned from the first native implementation of Satin (inside Manta) to a new, pure Java version, which is implemented on top of Ibis. Satin/Ibis exploits Java's "write once, run everywhere" feature: it can run on any JVM, and is therefore easy to use in a grid environment. Performance measurements show that, even on Fast Ethernet networks, high performance can be achieved, without the use of native code. Additional measurements on a real grid system with sites distributed over Europe show that CRS indeed outperforms RS by a wide margin. This confirms the simulation results presented in Chapter 6. Moreover, the fact that Satin/Ibis runs (without recompiling) on the widely different systems that are present in the testbed we used, validates our claim that a Java-centric solution greatly simplifies distributed supercomputing.

Satin/Ibis also has more functionality than the original Satin implementation in Manta. We described a *single* solution that solves two separate problems: it allows asynchronous exceptions from spawned methods to be handled, and makes it possible to express speculative parallelism in a straightforward way. This was achieved by offering a mechanism to execute a user-defined action when a spawned method is finished, and an abort primitive that retracts work that was speculatively spawned. Using this mechanism, a larger class of applications can be expressed in Satin, such as game-tree search. Because all network messages that are introduced with Satin/Ibis's inlet and abort mechanism are asynchronous, the implementation is also efficient in wide-area systems with high latencies.

Chapter 9

Conclusions

```
I believe that it is probably true that fortune is the arbiter of
half the things we do, leaving the other half or so to be
controlled by ourselves.

- Niccolo Machiavelli
```

In the introduction, we listed the contributions that this thesis makes. We repeat them here for convenience.

1. Using the Manta system, we show that object-based communication in Java, and in particular RMI, can be made highly efficient.

2. Using Ibis, we demonstrate that this is even possible while maintaining the portability and heterogeneity features (i.e., "write once, run everywhere") of Java.

3. We demonstrate that it is possible to write efficient, fine-grained distributed supercomputing applications with Java RMI, but that the programmer has to implement application-specific wide-area optimizations.

4. With Satin, we integrate divide-and-conquer primitives into Java, without changing the language.

5. We show that, using serialization on demand and user-level (zero-copy) communication, an efficient divide-and-conquer system that runs on distributed-memory machines and the grid can be implemented in Java.

6. We demonstrate that divide-and-conquer applications can be efficiently executed on hierarchical systems (e.g., the grid), without any wide-area optimizations by the application programmer, using novel wide-area-aware load-balancing algorithms.

7. We present a mechanism that is integrated into Java's exception handling model, and that allows divide-and-conquer applications to abort speculatively spawned work on distributed-memory machines.

8. We validate our claim that the Java-centric approach greatly simplifies the deploy-
ment of efficient parallel applications on the grid. We do this by running a parallel
application on a real grid testbed, using machines scattered over the whole of Eu-
rope.

Below, we will elaborate on the contributions we made, we will briefly summarize
the conclusions we presented in the separate thesis chapters, and evaluate the results we
achieved with this thesis. Finally, we give a brief overview of issues that are open for
future research.

9.1   RMI

In the first part of this thesis (Chapters 3 and 4), we investigated how to implement Java's
Remote Method Invocation efficiently, with the goal of using this flexible communication
mechanism for parallel programming. Reducing the overhead of RMI is more challeng-
ing than for other communication primitives, such as Remote Procedure Call (RPC), be-
cause RMI implementations must support inter operability and polymorphism. We have
designed new serialization and RMI implementations (Manta RMI) that support highly
efficient communication between machines that implement our protocol. Communication
with Java virtual machines (running the Sun RMI protocol) is also possible but slower.

We have demonstrated that RMI can be implemented almost as efficiently as Remote
Procedure Call, even on high-performance networks like Myrinet, while keeping the in-
herent advantages of RMI (polymorphism and inter operability). These results suggest
that an efficient RMI implementation is a good basis for writing high-performance paral-
lel applications.

Moreover, by comparing Manta RMI with the Sun compiled system, we demonstrated
that the Sun RMI *protocol* is inherently inefficient. It does not allow efficient implemen-
tations of RMI, because it enforces byte swapping, sends type information multiple times,
and makes a zero-copy implementation impossible.

We have also described our experiences in using Manta RMI on a geographically dis-
tributed (wide-area) system. The goal of this case study was to obtain actual experience
with a Java-centric approach to grid computing and to investigate the usefulness of RMI
for distributed supercomputing. The Java system we have built is highly transparent: it
provides a single communication primitive (RMI) to the user, even though the implemen-
tation uses several communication networks and protocols. In general, the RMI model
was easy to use on our wide-area system. To obtain good performance, the programs take
the hierarchical structure of the wide-area system into account and minimize the amount
of communication (RMIs) over the slow wide-area links. With such optimizations in
place, the programs can effectively use multiple clusters, even though they are connected
by slow links.

A problem is, however, that each application had to be optimized individually to re-
duce the utilization of the scarce wide-area bandwidth or to hide the large wide-area la-
tency. We found that different applications may require very different wide-area optimiza-
tions. In general, it is hard for a programmer to manually optimize parallel applications

for hierarchical systems. By comparing RMI with other programming models, we identified several shortcomings of the RMI model. In particular, the lack of asynchronous communication and of a broadcast mechanism complicates programming.

9.2   Satin

Ideally, the application programmer should not have to implement different wide-area optimizations for each application. One possible solution of this problem is investigated in detail in the second part of this thesis (Chapters 5, 6 and 8). We chose one specific class of problems, divide-and-conquer algorithms, and implemented an efficient platform consisting of a compiler and a runtime system that work together to apply wide-area optimizations automatically. We demonstrated that divide-and-conquer programming can be cleanly integrated into Java. The resulting system is called Satin. We showed that an efficient implementation of Satin on a cluster of workstations is possible by choosing convenient parameter semantics. An important optimization is the on-demand serialization of parameters to spawned method invocations.

Furthermore, in a grid environment, load balancing is an important factor for parallel applications. We described our experiences with five load-balancing algorithms for parallel divide-and-conquer applications on hierarchical wide-area systems. We showed that traditional load-balancing algorithms used with shared-memory systems and workstation networks achieve suboptimal results in a wide-area setting. Moreover, we demonstrated that hierarchical stealing, proposed in literature for load balancing in wide-area systems, performs even worse.

We introduced a novel load-balancing algorithm, called *Cluster-aware Random Stealing* (CRS). We used the Panda WAN emulator to evaluate the performance of Satin under many different WAN scenarios. Our experiments showed that Satin's CRS algorithm can actually tolerate a large variety of WAN link performance settings, and schedule parallel divide-and-conquer applications such that they run almost as fast on multiple clusters as they do on a single, large cluster. When CRS is modified to adapt to the performance of the WAN links, the performance of Satin on real-life scenarios (replayed NWS data) improves even more. We also verified our simulation results by running an application on a real wide-area system which contains machines located throughout Europe. These strong results suggest that divide-and-conquer parallelism is a useful model for writing distributed supercomputing applications on hierarchical wide-area systems.

We also described a *single* solution that solves two separate problems at once. It allows exceptions from spawned methods to be handled, and makes it possible to express speculative parallelism in a straightforward way. This was achieved by offering a mechanism to execute a user-defined action when a spawned method is finished, and an abort primitive that retracts work that was speculatively spawned. Because all network messages that are introduced with Satin/Ibis's inlet and abort mechanism are asynchronous, the implementation is also efficient in wide-area systems with high latencies.

9.3   Java-centric Grid Computing

In the first part of this thesis (Chapters 3–6), we have used the Manta system to investigate the usefulness of Java for grid computing, first for RMI and next for Satin. The fact that Manta uses a native compiler and runtime system allowed us to experiment with different programming models, implement and test several optimizations and to do detailed performance analysis. This would have been much more difficult with a JIT. However, since our solutions were integrated into a native Java system, the useful "write once, run everywhere" feature of Java is lost, while this feature was the reason to use Java for grid computing in the first place.

In the second part of this thesis, we reimplemented Manta's efficient serialization and communication mechanisms, but this time in pure Java. The resulting system, called Ibis, allows highly efficient, object-based communication, combined with Java's "run everywhere" portability, making it ideally suited for high-performance computing in grids. The portability layer we designed for Ibis (called the IPL) provides a single, efficient communication mechanism using streaming and zero-copy implementations. The mechanism is flexible, because it can be configured at run time using properties. Efficient serialization can be achieved by generating serialization code in Java, thus avoiding run time type inspection, and by using special, typed buffers to reduce type conversion and copying.

The Ibis strategy to achieving both performance and portability is to develop efficient solutions using standard techniques that work everywhere, supplemented with highly optimized but non standard solutions for increased performance in special cases. Exploiting these features, Ibis is a flexible, efficient Java-based grid programming environment that provides a convenient platform for (research on) grid computing.

As a test case, we studied an efficient RMI implementation that outperforms previous RMI implementations in Java, and achieves performance similar to Manta RMI, without using native code. The RMI implementation also provides efficient communication on gigabit networks like Myrinet, but then some native code is required.

We have also implemented Satin on top of Ibis, applying the lessons we learned from the first native implementation of Satin (inside Manta) to a new, pure-Java version. Satin on top of Ibis exploits Java's "run everywhere" feature: it can run on any JVM, and is therefore easy to use in a grid environment. Performance measurements show that, even on Fast Ethernet networks, high performance can be achieved on grid systems, without the use of native code. Deployment of a parallel application on a real grid system shows that the Java-centric approach is feasible.

9.4   Summary and Evaluation

As stated in the Chapter 1, the ultimate goal of the research in this thesis is *to provide an easy-to-use programming environment for medium and fine-grained distributed supercomputing on hierarchical heterogeneous grids*. As was summarized in Table 1.1, several difficult problems had to be solved to achieve this goal.

In an easy-to-use grid-programming environment, it should be straightforward to write, compile, and run parallel programs for a wide range of different platforms. More-

over, it should be possible to use multiple grid computing resources simultaneously, even though the architectures of the resources may be different. Our approach to deal with the heterogeneity of grid environments was to use a Java-centric solution, because of Java's "run everywhere" feature.

A problem with this approach was that, although recent Java just-in-time compilers have excellent sequential execution performance characteristics, Java's communication performance was still suboptimal. In this thesis, we have shown that the communication problem can be solved. First, using the Manta system, we have identified important serialization and communication bottlenecks. A prototype native RMI implementation inside Manta demonstrated that Java RMI can be almost as efficient as C-based RPCs, even while preserving the benefits of RMI, such as polymorphism.

Next, using Ibis, we have shown that object-based communication can also be made efficient in 100% Java, without sacrificing Java's "run everywhere" portability. We have demonstrated that this can be achieved by using efficient compiler-generated serialization, and optimized streaming protocols, that avoid data copies.

However, good communication performance is not the only requirement for a grid programming environment. An equally hard problem is related to the structure of the grid. Some communication channels in a grid environment have a high performance, while others have extremely high latencies and low bandwidths. There can be several orders of magnitude difference between local and wide-area communication performance. To achieve good performance, grid application codes have to deal with this problem.

An important assumption we make in this thesis is that grids are hierarchically structured. This means that supercomputers or clusters of workstations are connected by wide-area links. We have investigated two grid programming models that are able to exploit this assumption.

First, when the hierarchical structure of the grid is exported to the application, RMI can be used to implement application-specific wide-area optimizations. We found that reasonable performance can be achieved with this approach, depending on the communication requirements of the application.

Second, we have demonstrated that divide-and-conquer programs can be executed efficiently on hierarchical systems, but that special, cluster-aware load-balancing algorithms are needed. However, the wide-area optimizations are done inside the runtime system of the programming environment, and the complexity of programming for the grid can be completely hidden from the application programmer.

To summarize, using a Java-centric approach and with Java's communication performance problems solved, we were able to fulfill, at least for the important class of divide-and-conquer applications, our goal of providing an easy-to-use programming environment for medium and fine-grained distributed supercomputing on hierarchical, heterogeneous grids. The application programmer does not have to implement any special wide-area optimizations. In general, for non divide-and-conquer applications, an optimized RMI implementation in combination with application-specific wide-area optimizations can be used to write "run everywhere" parallel applications for the grid.

9.5   Open Issues

Although we have shown that the Java-centric approach to grid computing is feasible, some open issues that have to be addressed still remain. For instance, in the extremely complex grid environments, hardware and software errors are more likely to occur than in supercomputers or clusters of workstations. It is therefore important that grid software is able to handle errors. In the future, we intend to investigate support for fault tolerance in Satin. We believe that fault tolerance solutions designed specifically for divide-and-conquer systems can be more straightforward than a general solution. Because spawned methods do not have side effects, it is possible to restart a spawned method if the execution of the original job failed. However, if the spawned job that failed is high in the spawn hierarchy, much work may have to be redone. Therefore, we propose a hybrid scheme that combines checkpointing and the restarting of spawned work.

Another area where more research is needed is security issues with respect to distributed supercomputing applications. The security problem can be split into three parts. First, unwanted access of grid resources by grid applications must be avoided. Java already largely solves this problem, because Java applications can be "sandboxed". The Java virtual machine, that is trusted because it is provided by the grid resource and not by the application, can prevent all accesses to restricted resources. Second, the grid application should be secured from attacks from the host. For instance, the host might try to temper with the input set of the application, or the application executable itself. Third, communication inside a parallel grid application should be safeguarded from outside parties. This can be achieved by encrypting the applications communication channels. Especially in the second and third areas, more research is needed.

Although we provide a programming environment that allows efficient execution of divide-and-conquer programs on the grid without special application-specific wide-area optimizations, more research is needed into other, more general grid programming models. For instance, it is possible to optimize collective communication operations for grid environments as is done with MagPIe [89]. Similar optimizations can be implemented for replicated objects. These optimizations can also be applied to a Java-centric system such as Ibis. This way, Java's "run everywhere" feature can also be exploited with MPI-style applications, and programs that use replicated objects.

Bibliography

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] M. Alt, H. Bischof, and S. Gorlatch. Program Development for Computational Grids using Skeletons and Performance Prediction. *Parallel Processing Letters*, 12(2):157–174, 2002. World Scientific Publishing Company.

[3] G. R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49–90, 1991.

[4] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling Multithreaded Java Bytecode for Distributed Execution. In *Proceedings of Euro-Par 2000*, number 1900 in Lecture Notes in Computer Science (LNCS), pages 1039–1052, München, Germany, August 2000. Springer-Verlag.

[5] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a Heterogeneous Multi-user Environment. In *10th International Parallel Processing Symposium*, pages 218–224, Honolulu, Hawaii, April 1996.

[6] L. Arantes, P. Sens, and B. Folliot. The Impact of Caching in a Loosely-coupled Clustered Software DSM System. In *Proceedings of IEEE International Conference on Cluster Computing, (CLUSTER'2000)*, pages 27–34, Chemnitz, Germany, December 2000. IEEE Society Press.

[7] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 4–11, Aizu, Japan, September 1999.

[8] M. Backschat, A. Pfaffinger, and C. Zenger. Economic Based Dynamic Load Distribution in Large Workstation Networks. In *Proceedings of Euro-Par'96*, number 1124 in Lecture Notes in Computer Science (LNCS), pages 631–634. Springer-Verlag, 1996.

[9] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldara, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving Simulation for Network Research. Technical Report 99–702, University of Southern California, 1999.

[10] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. *Journal of Parallel and Distributed Computing*, 40(1):49–64, Februari 1997.

[11] H. Bal et al. The Distributed ASCI Supercomputer Project. *ACM Special Interest Group, Operating Systems Review*, 2000. Information available online at http://www.cs.vu.nl/das/.

[12] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, Februari 1998.

[13] H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, and R. F. Hofman. Optimizing Parallel Applications for Wide-Area Clusters. In *12th International Parallel Processing Symposium (IPPS'98)*, pages 784–790, Orlando, FL, April 1998.

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

[15] M. O. Bender and M. O. Rabin. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. *Theory of Computing Systems*, 35:289–304, 2002.

[16] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level Scheduling on Distributed Heterogeneous Networks. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'96)*, Pittsburgh, PA, November 1996. Online at http://www.supercomp.org.

[17] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, 1995.

[18] R. Bhoedjang, K. Verstoep, T. Rühl, H. Bal, and R. Hofman. Evaluating Design Alternatives for Reliable Communication on High-Speed Networks. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-9)*, pages 71–81, Cambridge, MA, November 2000.

[19] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[20] A. Bik, J. Villacis, and D. Gannon. Javar: A Prototype Java Restructuring Compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, November 1997.

[21] D. Blackston and T. Suel. Highly Portable and Efficient Implementations of Parallel Adaptive N-Body Methods. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'97)*, November 1997. Online at http://www.supercomp.org.

[22] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 207–216, Santa Barbara, CA, July 1995.

[23] 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, Santa Fe, New Mexico, November 1994.

[24] R. D. 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, CA, 1997.

[25] 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, Februari 1995.

[26] A. Bonhomme, A. L. Cox, and W. Zwaenepoel. Heterogeneity in ThreadMarks DSM system. Technical report, Rice University, September 1997.

[27] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–955, 1998.

[28] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proceedings of the 1997 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 214–224, Seattle, WA, June 1997.

[29] M. Bull, L. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for Scientific Applications. In *Proceedings of the ACM 2001 Java Grande/ISCOPE (International Symposium on Computing in Object-Oriented Parallel Environments) Conference*, pages 97–105, June 2001.

[30] M. d. Burger, T. Kielmann, and H. E. Bal. TopoMon: A Monitoring Tool for Grid Network Topology. In *International Conference on Computational Science*, volume 2330 of *Lecture Notes in Computer Science (LNCS)*, pages 558–567, April 2002.

[31] M. Burke, J.-D. Choi, S. Fink, D. Grove, M.Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, June 1999.

[32] D. Butenhof. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley Pub. Co., May 1997. ISBN: 0201633922.

[33] B. Carpenter, G. Fox, S. H. Ko, and S. Lim. Object Serialization for Marshalling Data in a Java Interface to MPI . In *Proceedings of the ACM 1999 Java Grande Conference*, pages 66–71, San Francisco, CA, June 1999.

[34] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. MPI for Java: Position Document and Draft API Specification. Technical Report JGF-TR-03, Java Grande Forum, November 1998.

[35] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries, Second Edition, Volume 1*. The Java Series. Addison-Wesley Pub. Co., second edition edition, November 1997. ISBN: 0-201-31002-3.

[36] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries, Second Edition, Volume 2*. The Java Series. Addison-Wesley Pub. Co., second edition edition, November 1997. ISBN: 0-201-31003-1.

[37] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries, Second Edition, Volume 3. Supplement for the Java 2 Platform, Standard Edition, v1.2*. The Java Series. Addison-Wesley Pub. Co., 1997. ISBN: 0-201-485524-1.

[38] C.-C. Chang and T. von Eicken. A Software Architecture for Zero-Copy RPC in Java. Technical Report 98-1708, Cornell University, September 1998.

[39] C.-C. Chang and T. von Eicken. Interfacing Java with the Virtual Interface Architecture. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 51–57, San Francisco, CA, June 1999.

[40] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, second edition edition, September 2001. ISBN: 0262531968.

[41] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'93)*, Portland, Oregon, November 1993. Online at http://www.supercomp.org.

[42] J. Darlington and M. Reeve. Alice: a Multi-processor Reduction Machine for the Parallel Evaluation of Applicative Languages. In *1st Conference on Functional Programming Languages and Computer Architecture*, pages 65–75, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981. ACM Press.

[43] J. Davis and G. Kendall. An Investigation, using Co-Evolution, to Evolve an Awari Player. In *Proceedings of Congress on Evolutionary Computation (CEC2002)*, pages 1408–1413, Hilton Hawaiian Village Hotel, Honolulu, Hawaii, May 2002. ISBN: 0-7803-7282-4.

[44] M. Dewanchand and R. Blankendaal. The usablility of Java for developing parallel applications. Master's thesis, Vrije Universiteit Amsterdam, August 1999. http://www.cs.vu.nl/manta.

[45] P. M. Dickens, P. Heidelberger, and D. M. Nicol. A Distributed Memory LAPSE: Parallel Simulation of Message-Passing Programs. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, pages 32–38, July 1994.

[46] E. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):430–439, September 1965.

[47] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6(1):53–68, March 1986.

[48] Edison Design Group. Compiler Front Ends for the OEM Market, May 2002. http://www.edg.com.

[49] T. v. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[50] Entropia: Distributed computing. http://www.entropia.com.

[51] D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems*, 12(1):53–66, May 1996.

[52] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-Area Implementation of the Message Passing Interface. *Parallel Computing*, 24(12–13):1735–1749, 1998.

[53] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing Multiple Communication Methods in High-Performance Networked Computing Systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[54] I. Foster, J. Geisler, and S. Tuecke. MPI on the I-WAY: A Wide-Area, Multimethod Implementation of the Message Passing Interface. In *Proceedings of the 1996 MPI Developers Conference*, pages 10–17. IEEE Computer Society Press, 1996.

[55] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'98)*, Orlando, FL, November 1998. Online at http://www.supercomp.org.

[56] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, Summer 1997.

[57] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing In-frastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998. ISBN: 1-55860-475-8.

[58] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *5th ACM Conference on Computer and Communication Security*, pages 83–92, San Francisco, CA, November 1998.

[59] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., San Fransisco, CA, 1994. ISBN: 1-55860-253-4.

[60] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns, and Practice*. Addison-Wesley Pub. Co., 1999. ISBN: 0-201-30955-6.

[61] B. Freisleben and T. Kielmann. Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs. *Computers and Artificial Intelligence*, 14(6):579–596, 1995.

[62] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 212–223, Montreal, Canada, June 1998.

[63] K. Gatlin. *Portable High Performance Programming via ArchitectureCognizant Divide-and-Conquer Algorithms*. PhD thesis, University of California, San Diego, September 2000.

[64] K. S. Gatlin and L. Carter. Architecture-cognizant Divide and Conquer Algorithms. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'99)*, Portland, Oregon, November 1999. University of California, San Diego, Computer Science and Engineering Department. Online at http://www.supercomp.org.

[65] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[66] V. Getov. MPI and Java-MPI: Contrasts and Comparisons of Low-Level Communication Performance. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'99)*, Portland, OR, November 1999. Online at http://www.supercomp.org.

[67] V. Getov, S. F. Hummel, and S. Mintchev. High-performance Parallel Programming in Java: Exploiting Native Libraries. *Concurrency: Practice and Experience*, 10(11–13):863–872, 1998.

[68] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley Pub. Co., second edition edition, 2000. ISBN: 0-201-31008-2.

[69] P. Gray and V. Sunderam. IceT: Distributed Computing and Java. *Concurrency: Practice and Experience*, 9(11):1161–1167, November 1997.

[70] A. Grimshaw and W. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[71] D. Hagimont and D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, The Lake District, England, September 1998.

[72] S. B. Hassen, H. Bal, and C. Jacobs. A Task and Data Parallel Programming Language based on Shared Objects. *ACM. Transactions on Programming Languages and Systems*, 20(6):1131–1170, November 1998.

[73] T. Haupt, E. Akarsu, G. Fox, A. Kalinichenko, K.-S. Kim, P. Sheethalnath, and C.-H. Youn. The Gateway System: Uniform Web Based Access to Remote Resources. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 1–7, San Francisco, CA, June 1999.

[74] C. A. Herrmann and C. Lengauer. HDC: A Higher-Order Language for Divide-and-Conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000.

[75] S. Hirano, Y. Yasu, and H. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java. In *ACM 1998 workshop on Java for High-performance network computing*, Februari 1998. Online at http://www.cs.ucsb.edu/conferences/java98/.

[76] Y. Hu, W. Yu, A. Cox, D. Wallach, and W. Zwaenepoel. Runtime Support for Distributed Sharing in Strongly Typed Languages. Technical report, Rice University, 1999. Online at http://www.cs.rice.edu/~willy/TreadMarks/papers.html.

[77] N. Hutchinson, L. Peterson, M. Abbott, and S. O'Malley. RPC in the x-Kernel: Evaluating New Design Techniques. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 91–101, Litchfield Park, AZ, December 1989.

[78] Intel's Philanthropic Peer-to-Peer Program for curing cancer. Information online at http://www.intel.com/cure.

[79] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 15–24, San Francisco, CA, June 1999.

[80] D. B. Johnson and W. Zwaenepoel. The Peregrine High-performance RPC System. *Software - Practice and Experience*, 23(2):201–221, 1993.

[81] K. Johnson, F. Kaashoek, and D. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Symposium on Operating Systems Principles 15*, pages 213–228, December 1995.

[82] S. P. Jones, editor. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, April 2003. ISBN: 0521826144. See http://haskell.org.

[83] G. Judd, M. Clement, Q. Snell, and V. Getov. Design Issues for Efficient Implemen-
     tation of MPI in Java. In *Proceedings of the ACM 1999 Java Grande Conference*,
     pages 58–65, San Francisco, CA, June 1999.

[84] M. Kaashoek, D. Engler, G. Ganger, H. Briceno, R. Hunt, D. Mazières, T. Pinck-
     ney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flex-
     ibility on Exokernel Systems. In *16th ACM Symposium on Operating Systems
     Principles*, pages 52–65, Saint Malo, France, 1997.

[85] K. Kaneda, K. Taura, and A. Yonezawa. Virtual private grid: A command shell
     for utilizing hundreds of machines efficiently. In *2nd IEEE/ACM International
     Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 212–219,
     Berlin, Germany, May 2002.

[86] V. Karamcheti and A. Chien. Concert - Efficient Runtime Support for Concurrent
     Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of
     the ACM/IEEE Conference on Supercomputing (SC'93)*, pages 598–607, Portland,
     Oregon, November 1993. Online at http://www.supercomp.org.

[87] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed
     Shared Memory on Standard Workstations and Operating Systems. In *Proceedings
     of the Winter 1994 Usenix Conference*, pages 115–131, San Francisco, CA, January
     1994.

[88] T. Kielmann, H. E. Bal, J. Maassen, R. V. v. Nieuwpoort, L. Eyraud, R. Hofman,
     and K. Verstoep. Programming environments for high-performance grid comput-
     ing: the albatross project. *Future Generation Computer Systems*, 18(8):1113–1125.

[89] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MAG-
     PIE: MPI's Collective Communication Operations for Clustered Wide Area Sys-
     tems. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel
     Programming (PPoPP'99)*, pages 131–140, Atlanta, GA, May 1999.

[90] A. Kishimoto and J. Schaeffer. Transposition Table Driven Work Scheduling in
     Distributed Game-Tree Search. In *Proceedings of Fifteenth Canadian Conference
     on Artificial Intelligence (AI'2002)*, volume 2338 of *Lecture Notes in Artificial
     Intelligence (LNAI)*, pages 56–68. Springer-Verlag, 2002.

[91] D. Knuth and R. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelli-
     gence*, 6(4):293–326, 1975.

[92] A. Krall and R. Grafl. CACAO -A 64 bit JavaVM Just-in-Time Compiler. *Concur-
     rency: Practice and Experience*, 9(11):1017–1030, November 1997.

[93] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and
     M. Ahamad. Efficient Implementations of Java Remote Method Invocation (RMI).
     In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies
     and Systems (COOTS'98)*, pages 19–35, Santa Fe, NM, April 1998.

[94] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions Software Eng.*, 17(7):725–730, July 1991.

[95] D. Kurzyniec and V. Sunderam. Efficient Cooperation Between Java and Native Codes –JNI Performance Benchmark. In *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPDA)*, Las Vegas, Nevada, June 2001.

[96] C. Laffra, D. Lorch, D. Streeter, F. Tip, and J. Field. Jikes Bytecode Toolkit. http://www.alphaworks.ibm.com/tech/jikesbt.

[97] K. Langendoen, R. Hofman, and H. Bal. Challenging Applications on Fast Networks. In *HPCA-4 High-Performance Computer Architecture*, pages 125–137, Februari 1998.

[98] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating Polling, Interrupts, and Thread Management. In *The 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pages 13–22, Annapolis, Maryland, Oct. 1996.

[99] P. Launay and J.-L. Pazat. The Do! project: Distributed Programming Using Java. In *First UK Workshop Java for High Performance Network Computing*, Southampton, September 1998.

[100] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 36–43, San Francisco, CA, June 2000.

[101] C. Lee. Grid RPC, Events and Messaging. Global Grid Forum, Advanced Programming Models Research Group (http://www.gridforum.org), September 2001.

[102] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A Grid Programming Primer. Global Grid Forum, Advanced Programming Models Research Group, submitted for review as a Global Grid Forum document (see http://www.gridforum.org), August 2001.

[103] S. Y. Lee. *Supporting Guarded and Nested Atomic Actions in Distributed Objects*. Master's thesis, University of California at Santa Barbara, July 1998.

[104] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC 1928: SOCKS protocol version 5, 1996.

[105] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley Pub. Co., second edition edition, 1999. ISBN: 0-201-43294-3.

[106] I. Lipkind, I. Pechtchanski, and V. Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. In *Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 447–460, Oct. 1999.

[107] J. Liu and D. M. Nicol. *DaSSF 3.1 User's Manual*, 2001. Available online at http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/.

[108] B. Lowekamp and A. Beguelin. ECO: Efficient Collective Operations for Communication on Heterogeneous Networks. In *International Parallel Processing Symposium*, pages 399–405, Honolulu, HI, 1996.

[109] J. Maassen. *Method Invocation Based Communication Models for Parallel Programming in Java*. PhD thesis, Dept. of Computer Science, Vrije Universiteit Amsterdam, June 2003.

[110] J. Maassen, T. Kielmann, and H. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *In proceedings of LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 1–6, Washington DC, March 2002.

[111] J. Maassen, T. Kielmann, and H. E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 13(8-9):681–712, 2001.

[112] J. Maassen, R. V. v. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[113] J. Maassen, R. V. v. Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, GA, May 1999.

[114] M. W. Macbeth, K. A. McGuigan, and P. J. Hatcher. Executing Java Threads in Parallel in a Distributed-Memory Environment. In *Proceedings CASCON'98*, pages 40–54, Missisauga, ON, 1998. Published by IBM Canada and the National Research Council of Canada.

[115] S. Marlow, S. L. P. Jones, A. Moran, and J. H. Reppy. Asynchronous Exceptions in Haskell. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 274–285, 2001.

[116] E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: a Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[117] D. Mosberger and L. Peterson. Making Paths Explicit in the Scout Operating System. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–168, 1996.

[118] MPI Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994. Version 1.1 at http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html.

[119] G. Muller and U. P. Schultz. Harissa: A Hybrid Approach to Java Execution. *IEEE Software*, 16(2):44–51, 1999.

[120] M. O. Neary and P. Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. In *Proceedings of the Joint ACM 2002 Java Grande - ISCOPE (International Symposium on Computing in Object-Oriented Parallel Environments) Conference*, pages 56–65, Seattle, November 2002.

[121] M. O. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0: Java-Based Parallel Computing on the Internet. In *Euro-Par 2000 Parallel Processing*, number 1900 in Lecture Notes in Computer Science (LNCS), pages 1231–1238, Munich, Germany, August 2000. Springer-Verlag.

[122] A. Nelisse, J. Maassen, T. Kielmann, and H. E. Bal. CCJ: Object-based Message Passing and Collective Communication in Java. *Concurrency and Computation: Practice and Experience*, 15(3–5):341–369, March 2003.

[123] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 153–159, San Francisco, CA, June 1999.

[124] R. V. v. Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Euro-Par 2000 Parallel Processing*, number 1900 in Lecture Notes in Computer Science (LNCS), pages 690–699, Munich, Germany, August 2000. Springer-Verlag.

[125] R. V. v. Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In *Proceedings Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 34–43, Snowbird, UT, June 2001.

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

[127] R. V. v. Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Programming using the Remote Method Invocation Model. *Concurrency: Practice and Experience*, 12(8):643–666, 2000.

[128] R. V. v. Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.

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

[130] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[131] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, pages 1225–1242, November 1997.

[132] A. Plaat, H. E. Bal, and R. F. Hofman. Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects. In *Proceedings of High Performance Computer Architecture (HPCA-5)*, pages 244–253, Orlando, FL, January 1999.

[133] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence*, 87(1-2):255–293, November 1996.

[134] M. A. Poletto. *Language and Compiler Support for Dynamic Code Generation*. PhD thesis, Massachusetts Institute of Technology, September 1999.

[135] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for applications –a Way Ahead of Time (WAT) Compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 41–53, Portland, OR, 1997. USENIX Association Press.

[136] K. Raptis, D. Spinellis, and S. Katsikas. Java as Distributed Object Glue. In *World Computer Congress 2000*, Beijing, China, August 2000. International Federation for Information Processing.

[137] K. v. Reeuwijk, A. van Gemund, and H. Sips. Spar: a Programming Language for Semi-automatic Compilation of Parallel Programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, August 1997.

[138] A. Reinefeld. An Improvement of the Scout Tree-Search Algorithm. *Journal of the International Computer Chess Association*, 6(4):4–14, 1983.

[139] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Romke, and J. Simon. The MOL Project: An Open Extensible Metacomputer. In *Proceedings of the Heterogenous computing workshop HCW'97 at IPPS'97*, pages 17–31. IEEE Computer Society Press, April 1997.

[140] R. v. Renesse, J. van Staveren, and A. Tanenbaum. Performance of the Amoeba Distributed Operating System. *Software — Practice and Experience*, 19:223–234, March 1989.

[141] C. Riley, S. Chatterjee, and R. Biswas. High-Performance Java Codes for Computational Fluid Dynamics. In *Proceedings of the ACM 2001 Java Grande/ISCOPE (International Symposium on Computing in Object-Oriented Parallel Environments) Conference*, pages 143–152, June 2001.

[142] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):28–38, June 1993.

[143] S. Rodrigues, T. Anderson, and D. Culler. High-Performance Local Communication With Fast Sockets. In *Proc. of the USENIX Annual Technical Conference '97*, pages 257–274, Anaheim, CA, January 1997.

[144] J. W. Romein and H. E. Bal. Wide-Area Transposition-Driven Scheduling. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 347–355, San Francisco, CA, August 2001.

[145] J. W. Romein and H. E. Bal. Awari is Solved (note). *Journal of the International Computer Games Association (ICGA)*, 25(3):162–165, September 2002.

[146] R. Rugina and M. Rinard. Automatic Parallelization of Divide and Conquer Algorithms. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 72–83, Atlanta, May 1999.

[147] T. Rühl, H. E. Bal, G. Benson, R. A. F. Bhoedjang, and K. Langendoen. Experience with a Portability Layer for Implementing Parallel Programming Systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 1477–1488, Sunnyvale, CA, 1996.

[148] G. Sampemane, L. Rivera, L. Zhang, and S. Krishnamurthy. HP-RMI : High Performance Java RMI over FM. University of Illinois at Urbana-Champaign. Online at http://www-csag.ucsd.edu/individual/achien/cs491-f97/projects/hprmi.html, 1997.

[149] L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, 15(5/6), 1999.

[150] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure. In *HPCN Europe*, pages 491–502, 1997.

[151] M. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8:1–17, Februari 1990.

[152] N. G. Shivaratri, P. Krueger, and M. Ginghal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, 25(12):33–44, December 92.

[153] D. Slate and L. Atkin. Chess 4.5 –The Northwestern Univeristy Chess Program. *Chess Skill in Man and Machine*, pages 82–118, 1977.

[154] L. Smarr and C. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[155] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: a Scientific Tool for Modeling Computational Grids. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'2000)*, Dallas, TX, November 2000. Online at http://www.supercomp.org.

[156] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on Project Serendip data and 100,000 personal computers. In C. Cosmovici, S. Bowyer, and D. Werthimer, editors, *Astronomical and Biochemical Origins and the Search for Life in the Universe", Proceedings of the Fifth Intl. Conf. on Bioastronomy*, number 161 in IAU Colloq. Editrice Compositori, Bologna, Italy, 1997. http://setiathome.ssl.berkeley.edu/.

[157] Sun Microsystems. *Java Remote Method Invocation Specification*, jdk 1.1 fcs, online at http://java.sun.com/products/jdk/rmi edition, Februari 1997.

[158] I. Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2, June 1988. RFC 1057.

[159] Sun MicroSystems, Inc. Java Object Serialization Specification, 1996. ftp://ftp.javasoft.com/docs/jdk1.1/serial-spec.ps.

[160] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, 1990.

[161] D. Thain, T. Tannenbaum, and M. Livny. *Grid Computing: Making The Global Infrastructure a Reality*, chapter "Condor and the Grid". John Wiley & Sons, Ltd., 2003. ISBN: 0-470-85319-0.

[162] C. Thekkath and H. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[163] E. Thieme. Parallel Programming in Java: Porting a Distributed Barnes-Hut implementation. Master's thesis, Vrije Universiteit Amsterdam, August 1999. http://www.cs.vu.nl/manta.

[164] B. Topol, M. Ahamad, and J. T. Stasko. Robust State Sharing for Wide Area Distributed Applications. In *18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 554–561, Amsterdam, The Netherlands, May 1998.

[165] R. Veldema. *Compiler And Runtime Optimizations for Fine-Grained Distributed Shared Memory Systems*. PhD thesis, Dept. of Computer Science, Vrije Universiteit Amsterdam, Oktober 2003.

[166] W. G. Vree. *Design Considerations for a Parallel Reduction Machine*. PhD thesis, Dept. of Computer Science, University of Amsterdam, 1989.

[167] J. Waldo. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency*, pages 5–7, July 1998.

[168] D. Wallach, W. Hsieh, K. Johnson, M. Kaashoek, and W. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proceedings 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 217–226, Santa Barbara, CA, July 1995.

[169] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: a Parallel Architecture for Declarative Programming. In *15th IEEE/ACM Symposium on Computer Architecture*, pages 124–130, Honolulu, Hawaii, 1988.

[170] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O from Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.

[171] A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, 17(3):44–53, May 1997.

[172] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running EveryWare on the Computational Grid. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'99)*, Portland, Oregon, November 1999. Online at http://www.supercomp.org.

[173] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[174] I.-C. Wu and H. Kung. Communication Complexity for Parallel Divide-and-Conquer. In *32nd Annual Symposium on Foundations of Computer Science (FOCS '91)*, pages 151–162, San Juan, Puerto Rico, Oct. 1991.

[175] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a High-performance Java Dialect. In *ACM 1998 workshop on Java for High-performance network computing*, pages 1–13, New York, NY, Februari 1998. ACM Press.

[176] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.

[177] S. Zhou, S. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.

Samenvatting

```
Hoe meer zand uit de zandloper van ons leven ontsnapt is,
hoe beter we erdoorheen zouden moeten kunnen kijken.

- Niccolo Machiavelli
```

Grid computing is een relatief nieuw vakgebied dat tot doel heeft om computersystemen over de hele wereld aan elkaar te koppelen en te integreren zodat zij gebruikt kunnen worden als één enkel systeem. Zo'n systeem moet het bijvoorbeeld mogelijk maken om grootschalige parallelle programma's tegelijkertijd over de hele wereld te draaien. Dit proefschrift kijkt vooral naar een specifieke vorm van grid computing, namelijk het gebied van gedistribueerd parallel rekenen, waarbij computers die over de hele wereld verspreid staan, samenwerken om één probleem op te lossen. De verzameling systemen waarop het programma draait wordt vaak een *grid* genoemd. Het doel is om de rekenkracht van de systemen samen te voegen om zo de oplossing van het uitgeprogrammeerde probleem sneller te vinden. Het bekendste voorbeeld van zo'n toepassing is misschien wel SETI@home, een programma waarmee iedereen thuis de hemel kan afspeuren naar buitenaardse intelligentie.

Dit proefschrift met de titel *"Efficiënt Java geöriënteerd Rekenen in een Gridomgeving"* presenteert onderzoek naar programmeeromgevingen die het schrijven van programmatuur voor gridomgevingen proberen te vergemakkelijken. Het is belangrijk om te realiseren dat het doel van dit alles is om problemen sneller op te lossen: de programmeeromgeving moet dus vooral efficiënt draaien.

Het is erg moeilijk om programma's voor een gridomgeving te schrijven en om die daarop te draaien, omdat grids van nature enorm heterogeen zijn, dat wil zeggen dat de onderliggende systemen in een grid verschillende processoren, besturingssystemen en programmeerbibliotheken gebruiken. Toch moeten deze verschillende systemen samenwerken om het programma te draaien. Met de bestaande programmeertechnieken is dat wel te realiseren, maar dit vergt een enorme discipline van de programmeur. Dit proefschrift onderzoekt of het mogelijk is om een programmeeromgeving voor grids te ontwerpen die puur op de programmeertaal Java gebaseerd is. Java heeft een voordeel ten opzichte van traditionele programmeertalen zoals C, C++ en Fortran, namelijk dat het gebruik maakt van een tussenstap: de virtuele machine oftewel *JVM*. Deze virtuele machine abstraheert van de onderliggende machine en het besturingssysteem en biedt de Java programmeur een uniforme programmeeromgeving. Hoofdstuk 2 van dit proefschrift geeft

achtergrondinformatie over Java en over de applicaties die we gebruiken.

Een probleem met het gebruik van Java was dat de executiesnelheid achterbleef bij die van traditionele programmeertalen. Dit probleem is echter recentelijk opgelost met de komst van efficiënte Just-In-Time (net op tijd) compilers, die de Java code tijdens het draaien van het programma vertalen naar machine instructies. Een hardnekkig probleem blijft over: de lage communicatiesnelheid van Java. Dit proefschrift verschaft oplossingen voor dit laatste probleem.

Java biedt een standaardoplossing voor communicatie tussen verschillende systemen, namelijk het aanroepen van procedures die op een andere machine uitgevoerd moeten worden. Dit heet in Java "Remote Method Invocation" of RMI. Deze procedures kunnen willekeurig ingewikkelde datastructuren mee krijgen als parameters en kunnen deze ook retourneren. Voordat Java datastructuren verstuurd kunnen worden, moeten ze eerst "platgeslagen" en geconverteerd worden naar een formaat waar de communicatie component iets mee kan. Dit proces heet *serialisatie*. RMI maakt gebruik van Java's serialisatie-mechanisme alvorens de data te versturen. Serialisatie en communicatie hangen dan ook nauw samen.

In hoofdstuk 3 van dit proefschrift laten we zien dat serialisatie en RMI niet alleen zeer inefficiënt geïmplementeerd zijn in de bestaande Java omgevingen, maar ook dat het ontwerp van de gebruikte protocollen inherent inefficiënt is, omdat het dataconversie en onnodig kopiëren van gegevens afdwingt. Om dit te laten zien hebben we een Java systeem ontwikkeld, *Manta* genaamd, dat onder andere de standaard serialisatie- en RMI-implementaties bevat. Het feit dat Manta Java code direct naar machine-instructies vertaalt zonder de tussenstap van de virtuele machine, stelt ons in staat om een gedetailleerde analyse te maken van de (in)efficiëntie van serialisatie, RMI en de onderliggende protocollen. Manta bevat tevens alternatieve, geoptimaliseerde implementaties van zowel serialisatie als RMI, waarmee we laten zien hoe communicatie in Java wel efficiënt gemaakt kan worden, zelfs als gigabit-per-seconde netwerken zoals Myrinet gebruikt worden. Deze snelle RMI implementatie is voor een procedureaanroep zonder parameters via Myrinet ongeveer een factor 35 sneller dan de standaardimplementatie.

Een belangrijke aanname die we in dit proefschrift doen is dat grids *hiërarchische systemen* zijn. Dat wil zeggen dat er niet bijvoorbeeld één losse computer in Delft met één losse computer in Amsterdam wordt verbonden, maar dat hele groepen van computers worden verbonden door lange-afstand netwerken. Zulke groepen worden dan "clusters" genoemd. Een goed voorbeeld van een hiërarchisch systeem is de DAS-2, een Nederlands systeem dat bestaat uit vijf clusters die bij verschillende Nederlandse universiteiten staan. Deze clusters zijn verbonden via lange-afstand netwerken. Omdat een enkele machine in DAS-2 ook al twee processoren heeft, heeft de hiërarchie in dit geval zelfs drie lagen.

Nu het probleem van Java's communicatiesnelheid is opgelost, is het mogelijk om Java te gebruiken om gedistribueerde, parallelle programma's te schrijven. Dit proefschrift onderzoekt twee verschillende methodes om zulke programma's in Java voor (hiërarchische) gridomgevingen te schrijven.

Ten eerste bestuderen we in hoofdstuk 4 of RMI gebruikt kan worden om parallelle applicaties te schrijven, die zelfs nog efficiënt draaien als de systemen wereldwijd verspreid zijn. Het blijkt dat het wel degelijk mogelijk is om efficiënt op een wereldwijd systeem te draaien, maar dat zeer uiteenlopende optimalisaties per applicatie nodig zijn.

Helaas is het uitdenken en implementeren van applicatie-specifieke optimalisaties niet erg programmeur-vriendelijk.

De tweede methode om parallelle applicaties te schijven voor gridomgevingen die dit proefschrift onderzoekt, spitst zich toe op een bepaalde klasse van applicaties, namelijk applicaties die de zogenaamde "verdeel en heers" strategie gebruiken. Dit zijn applicaties die een probleem opsplitsen in deelproblemen, totdat het werk zover opgesplitst is dat het eenvoudig uitgevoerd kan worden. Tenslotte worden alle deeloplossingen gecombineerd tot het uiteindelijke resultaat. Hoofdstuk 5 presenteert *Satin*, een programmeeromgeving die het gemakkelijk maakt om zulke "verdeel en heers" programma's te schrijven. De programmeur zelf hoeft met Satin geen enkele optimalisatie voor gridsystemen te implementeren. De Satin taal is gebaseerd op Java en de implementatie die we gebruiken in hoofdstuk 5 maakt gebruik van het eerder genoemde Manta systeem. Hierdoor is het mogelijk om gedetailleerde metingen te doen naar de efficiëntie van het Satin systeem. "Verdeel en heers" programma's genereren erg veel taken van zeer uiteenlopende grootte, die parallel uitgevoerd kunnen worden. Een belangrijke optimalisatie die geïntroduceerd wordt in dit proefschrift is het idee dat de gegevens die bij de parallelle taken van Satin programma's horen, pas geserialiseerd worden op het moment dat ze daadwerkelijk het netwerk overgestuurd gaan worden en niet op het moment dat de taak wordt aangemaakt. Dit verbetert de performance van het parallelle programma enorm, omdat het blijkt dat het overgrote deel van de taken wordt uitgevoerd op de machine die ze gegenereerd heeft.

Terwijl hoofdstuk 5 alleen naar de efficiëntie kijkt van Satin programma's die draaien op één machine of op één enkele cluster, bestudeert hoofdstuk 6 het draaien van Satin programma's op wereldwijde systemen. Eerst demonstreren we dat traditionele algoritmes om de taken van "verdeel en heers" programma's te verdelen over de verschillende machines slecht functioneren op wereldwijde systemen.

Hiërarchische algoritmes leggen een boomstructuur over de machines heen. Berichten worden altijd via de boomstructuur verzonden, waardoor de trage communicatie tussen clusters tot een minimum gereduceerd wordt. We laten zien dat de algemene veronderstelling dat er hiërarchische algoritmes nodig zijn in gridomgevingen, onjuist is. De communicatie wordt weliswaar tot een minimum beperkt, maar de hoeveelheid werk wordt onevenredig verdeeld, waardoor sommige machines lange tijd niets te doen hebben, terwijl andere machines overladen worden met werk.

Als oplossing presenteert hoofdstuk 6 een nieuw algoritme, *CRS* genaamd, dat wel uitstekende prestaties levert op wereldwijde systemen. CRS is gebaseerd op het "stelen" van werk vanaf willekeurige machines in het systeem. Een gedetailleerde performance analyse, mogelijk gemaakt omdat Satin op Manta is gebaseerd, laat zien dat CRS op een wereldwijd systeem, voor 11 van de 12 applicaties die we getest hebben, ten hoogste 4% verliest ten opzichte van een enkele cluster met evenveel machines. CRS werkt zelfs goed met extreem langzame netwerkverbindingen tussen de clusters van het systeem.

In een enkel zéér extreem scenario waarbij sommige verbindingen zeer snel zijn en andere juist zeer langzaam, laat de prestatie van CRS nog iets te wensen over. Hoofdstuk 6 presenteert nog een nieuw algoritme, *ACRS*, dat wel goed presteert in deze extreme gevallen. Het verschil tussen CRS en ACRS is dat de laatste de parameters van de kansverdeling, die bepaalt bij welke machine gestolen wordt, aanpast aan de huidige netwerk performance en vaker werkt steelt van machines met een snelle netwerkverbinding.

De hoofdstukken 4 tot en met 6 tezamen maken duidelijk dat het draaien van parallelle applicaties op wereldwijde systemen met behulp van Java zeer wel mogelijk is en dat deze zelfs relatief eenvoudig te programmeren zijn. Zeker in Satin is dit het geval, omdat de programmeur daar helemaal geen gridoptimalisaties hoeft te schrijven. Echter, omdat Manta (en dus ook Satin) Java programma's direct naar machinecode vertaalt, gaat het portabiliteitsvoordeel van Java verloren. In de hoofdstukken 7 en 8 lossen we dit probleem op door de lessen die we met Manta geleerd hebben opnieuw toe te passen, maar nu in puur Java. Het resulterende systeem moet op elke standaard JVM draaien.

Hoofdstuk 7 laat zien hoe het mogelijk is om efficiënte serialisatie en communicatie te implementeren in puur Java. De architectuur die we hiervoor gebruiken heet *Ibis*. Het voordeel van Ibis ten opzichte van Manta, is dat het op elk willekeurig platform draait, zolang er maar een JVM voor beschikbaar is. Ibis kan gebruik maken van verschillende types netwerken en biedt daarvoor één enkel, eenvoudig te gebruiken programmeermodel voor aan. Om Ibis uit te testen hebben we RMI opnieuw geïmplementeerd op Ibis, weer in puur Java. Het blijkt dat deze nieuwe implementatie tot een factor 10 sneller is dan de standaard Java implementatie. Toch draait de Ibis RMI versie gewoon op elk Java systeem.

In Hoofdstuk 8 implementeren we Satin opnieuw, nu met behulp van Ibis. Ook dit blijkt zeer efficiënt te kunnen. De Satin versie op Ibis bevat ook een nieuw mechanisme dat het mogelijk maakt om speculatief werk op te starten en dit werk later weer af te schieten als het achteraf toch overbodig blijkt te zijn. Deze nuttige uitbreiding is naadloos geïntegreerd in Java's foutafhandelingsmechanisme. Met behulp van de nieuwe uitbreiding is het mogelijk om een grotere klasse van applicaties uit te drukken in Satin, zoals bijvoorbeeld efficiënte zoekalgoritmes. Om het nieuwe mechanisme uit te testen gebruiken we het spel Awari. Het blijkt dat Awari veel efficiënter werkt nu Satin speculatief parallellisme ondersteunt, terwijl het nog steeds eenvoudig op te schrijven is.

Tenslotte laten we zien dat Satin ook in de praktijk werkt, door een applicatie (een raytracer) te draaien op het GridLab testbed. Dit laatste is een verzameling zeer verschillende machines die over heel Europa verspreid staan. Het mooie van de Java-gebaseerde oplossing is dat het programma slechts één keer gecompileerd hoefde te worden, naar code voor Java's virtuele machine. Vervolgens was het mogelijk om, zonder ook maar een letter van het programma aan te passen, de applicatie te draaien op alle verschillende architecturen en besturingssystemen die aanwezig waren in het testbed. Dit is een goed voorbeeld van het voordeel van Java ten opzichte van traditionele programmeersystemen, waarbij het noodzakelijk zou zijn om het programma voor elke architectuur opnieuw te compileren en mogelijk zelfs aan te passen voor elk nieuw systeem.

Curriculum Vitae

Personal data

Name:	Robert Vincent van Nieuwpoort
Date of birth:	May 30th, 1975, Alphen aan den Rijn, The Netherlands
Nationality:	Dutch

Current address: Vrije Universiteit
Faculty of Sciences
Department of Computer Science
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
rob@cs.vu.nl

Education

1987-1993   VWO Gymnasium
Herbert Vissers College
Nieuw-Vennep, The Netherlands

1993-1998   Master's degree in Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

1998-2002   Ph.D student
Vrije Universiteit
Amsterdam, The Netherlands

Professional Experience

1997-1998	Teaching assistant for the courses *Introduction to Programming*, *Data-structures*, *Software Engineering* and *Assembly Programming* Vrije Universiteit Amsterdam, The Netherlands
1999-2000	Teacher for the lecture *Computer Organization and Operating Systems* Vrije Universiteit Amsterdam, The Netherlands
2001-2003	Teaching assistant for the course *Parallel Programming* Vrije Universiteit Amsterdam, The Netherlands
2002-present	Researcher Vrije Universiteit Amsterdam, The Netherlands

Publications

Journal Publications

- Rob V. van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald S. Veldema. Wide-area Parallel Programming Using the Remote Method Invocation Model. In *Concurrency Practice and Experience*, (Special Issue: ACM 1999 Java Grande Conference (Part 3). Issue Edited by Geoffrey Fox.) volume 12, issue 8, pages 643–666, Online ISSN: 1096-9128, Print ISSN: 1040-3108, July 2000.

- Jason Maassen, Rob V. van Nieuwpoort, Ronald S. Veldema, Henri E. Bal, Thilo Kielmann, Ceriel J.H. Jacobs, and Rutger F.H. Hofman. Efficient RMI for Parallel Programming. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 23, number 6, pages 747–775, November 2001.

- Thilo Kielmann, Henri E. Bal, Jason Maassen, Rob V. van Nieuwpoort, Lionel Eyraud, Rutger F.H. Hofman, and Kees Verstoep. Programming Environments for High-Performance Grid Computing: the Albatross Project. In *Future Generation Computer Systems*, volume 18, number 8, pages 1113–1125, 2002.

- Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger F.H. Hofman, Ceriel J.H. Jacobs, Thilo Kielmann, Henri E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment.
  Accepted for publication in *Concurrency and Computation, Practice and Experience* (Special Issue Java Grande/ISCOPE 2002).

Conference Publications

- Jason Maassen, Rob V. van Nieuwpoort, Ronald S. Veldema, Henri E. Bal, and Aske Plaat. An Efficient Implementation of Java's Remote Method Invocation. In Proceedings of the *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, Georgia, May 4–6, 1999.

- Henri E. Bal, Aske Plaat, Thilo Kielmann, Jason Maassen, Rob V. van Nieuwpoort, and Ronald S. Veldema. Parallel Computing on Wide-Area Clusters: the Albatross Project. In Proceedings of the *Extreme Linux Workshop*, pages 20–24, Monterey, California, June 8–10, 1999.

- Rob V. van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald S. Veldema. Wide-area parallel computing in Java. In Proceedings of the *ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, California, June 12–14, 1999.

- Thilo Kielmann, Henri E. Bal, Jason Maassen, Rob V. van Nieuwpoort, Ronald S. Veldema, Rutger F.H. Hofman, Ceriel J.H. Jacobs, and Kees Verstoep. The Albatross Project: Parallel Application Support for Computational Grids. In Proceedings of the *1st European GRID Forum Workshop*, pages 341–348, Poznan, Poland, April 12–13, 2000.

- Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In Proceedings of *Euro-Par 2000 Parallel Processing*, number 1900 in Lecture Notes in Computer Science (LNCS), pages 690–699, Munich, Germany, August 29–September 1, 2000.

- Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In Proceedings of the *Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 34–43, Snowbird, Utah, June 18–19, 2001.

- Rob V. van Nieuwpoort, Jason Maassen, Rutger F.H. Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In Proceedings of the *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, November 3–5, 2002.

Other Publications

- Ronald S. Veldema, Rob V. van Nieuwpoort, Jason Maassen, Henri E. Bal, and Aske Plaat. Efficient Remote Method Invocation. *Technical Report IR-450*, Vrije Universiteit Amsterdam, September 1998.

- Rob V. van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald S. Veldema. Wide-area parallel computing in Java. In *ASCI'99, Proceedings of the Fifth Annual Conference of the Advanced School for Computing and Imaging*, pages 338–347, Heijen, The Netherlands, June 15–17, 1999.

- Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *ASCI 2000, Proceedings of the Sixth Annual Conference of the Advanced School for Computing and Imaging*, pages 177–184, Lommel, Belgium, June 14–16, 2000.

- Henri E. Bal et al. The distributed ASCI supercomputer project. In *ACM Special Interest Group, Operating Systems Review*, volume 34, number 4, pages 76–96, October 2000.

- Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In *ASCI 2001, Proceedings of the Seventh Annual Conference of the Advanced School for Computing and Imaging*, pages 459–466, Heijen, The Netherlands, May 30–June 1, 2001.

- Rob V. van Nieuwpoort, Jason Maassen, Rutger F.H. Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *ASCI 2003, Proceedings of the Ninth Annual Conference of the Advanced School for Computing and Imaging*, pages 208–215, Heijen, The Netherlands, June 4–6, 2003.

Index

SPIN, 68
Split-C, 68
stub, 24
Successive Over-Relaxation (SOR), 31
Sun compiled RMI, 41, 56
Sun RMI, 41
SuperWeb, 16
synchronized modifier and keyword, 20
synchronous communication, 85

TCP/IP (Transmission Control Protocol/Internet Pro-
        tocol), 6, 7, 18, 46, 62, 72, 148, 151,
        154, 156, 158, 165
thread pool, 53
threads, 6, 19, 46, 51, 57, 74, 77, 85, 91, 153, 165,
        181, 201, 210
Titanium, 27
TopoMon, 171
traceroute, 200
transient, 22, 93
transposition table, 205
Transposition-Driven Scheduling (TDS), 87
Traveling Salesperson (TSP), 33
TreadMarks, 26

Unicode, 63, 159
upcall, 6, 60, 86, 153, 166–168
User Datagram Protocol (UDP), 7, 18, 51, 53, 67,
        156, 163, 166
UTF-8 format, 63, 159

VIA, 68
virtual method table (or vtable), 48
VJava, 67

wait/notify construct, 21
water, 32
Wide Area Network (WAN), 7
wide-area network emulator, 119

zero-copy, 53, 149