Scalability and Fault Tolerance in Global Computing

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Michael O. Neary

Committee in charge:

Professor Peter Cappello, Chair
Professor Klaus E. Schausser
Professor Richard Wolski

June 2002
The dissertation of Michael O. Neary is approved:

Chair

June 2002
Scalability and Fault Tolerance in Global Computing

Copyright 2002

by

Michael O. Neary
To my dear grandmother,

Hilde Deybrich,

who traveled to a foreign country for the first time in her life,

and celebrated her 85th birthday on my graduation day.
Acknowledgements

First, I want to take this opportunity to thank my advisor, Professor Peter Cappello, for his invaluable advice and guidance on conducting this research, and his patience in so many discussions.

I also want to thank my former advisor, Professor Klaus Schausser. Without his help, I would not have had the opportunity of coming to UCSB. In addition, I would not have been able to graduate without his continuing financial support of my research. And, most importantly, it was him who had the original idea for this project.

Next, I want to thank Professor Anurag Acharya, who was on my committee for over three years. His critical and highly constructive comments at the time of my qualifying exam helped to shape the direction of this research. His financial support during the summer of 1999 was much appreciated.

Special thanks goes to Professor Rich Wolski for stepping onto my committee at short notice, and providing valuable comments in preparation of my defense.

Over time, the following students contributed significantly to this project: Bernd Oliver Christiansen, who designed and implemented the initial Javelin prototype; Paul Kmiec, who implemented the Javelin RMI Class Loader; Alan
Phipps, who contributed the Broker Name Service implementation; Steven Richman, who implemented an alternate branch-and-bound Eager Scheduler, and compared his version to mine; and Sylvie Dieckmann, who implemented the initial Kruskal Minimum Spanning Tree optimization to the TSP application. Many thanks to all of you!

Obtaining the final results of this dissertation would not have been possible without the free, unlimited access to the Linux cluster at University of Paderborn, Germany. Special thanks go to Dr. Joern Gehring for getting me the account, and Andreas Krawinkel and the other system administrators at the Paderborn Center for Parallel Computing for their fabulous support.

I would also like to thank the Department of Computer Science in general, and the students of the Parallel Systems Lab in particular, for creating such a fun work environment.

Finally, I would like to thank my family back home in Germany, and my friends on both sides of the Atlantic for their strong support and encouragement throughout these years — without you all, I could not have done it!
Curriculum Vitæ

Michael O. Neary

Personal

Name           Michael Oliver Neary
Date of Birth  July 12, 1966
Place of Birth Bielefeld, Germany
Nationalities  Germany and United Kingdom
Email          neary@cs.ucsb.edu

Education

1987–1995      Diplom (M.S.) in Computer Science, University of Paderborn, Paderborn, Germany.

Conference & Journal Publications:


- Michael O. Neary and Peter Cappello. Internet-Based TSP Computation with Javelin++. 2000 International Conference on Parallel Processing,


- Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O.


Abstract

Scalability and Fault Tolerance in Global Computing

by

Michael O. Neary

Javelin 3 is a Java-based software system for scalable, fault tolerant, adaptively parallel “Global” (a.k.a. Internet or Grid) Computing. Projects like SETI@home have recently garnered a lot of popular interest in this field; however, they are largely geared to specific applications. Javelin 3 is intended to free application developers from concerns about complex inter-processor communication, task scheduling, and fault tolerance among networked hosts. When all or part of their application can be cast as a master-worker or a branch-and-bound computation, Javelin 3 allows developers to focus on the underlying application. The dissertation highlights the scalability and fault tolerance of Javelin 3, and the distributed work stealing and advanced eager scheduling mechanisms used. The scheduling strategy enables dynamic task decomposition, which improves load balancing in the presence of tasks whose non-uniform computational load is evident only at execution time.
We provide an analysis of the expected performance degradation due to unresponsive hosts, and measurements of actual performance degradation due to unresponsive hosts. We also present speedup measurements of a large-scale branch-and-bound application, using up to 1,024 hosts.
Contents

1 Introduction 1

2 Related Work 13
   2.1 Standalone Applications ................................ 14
   2.2 Programming Models ........................................ 16
   2.3 Java-Based Network Computing Projects .................. 18
      2.3.1 Adaptively Parallel Systems ....................... 18
      2.3.2 Traditional Parallel Programming Systems ....... 21
   2.4 Grid Computing Projects ................................. 23
   2.5 Evaluation .............................................. 25

3 System Architecture 31
   3.1 Overview .................................................. 32
   3.2 Javelin Broker Name Service ............................... 35
   3.3 Broker Network & Host Tree ............................... 37
   3.4 Code Distribution ........................................ 39

4 Scalable Computation 42
   4.1 The Work Stealing Scheduler .............................. 42
   4.2 Shared Memory for Bound Propagation .................. 45
   4.3 Asynchronous Communication Model ..................... 47

5 Advanced Eager Scheduling 49
   5.1 The Master-Worker Eager Scheduler ..................... 50
   5.2 The Branch-and-Bound Eager Scheduler .................. 54
      5.2.1 Deferred Detection of Pruned Nodes ............... 58
List of Figures

3.1 The Javelin 3 Architecture. .......................... 33
3.2 State Transition Diagram for Javelin 3 Hosts. ........ 35
3.3 JavelinBNS Lookup Sequence. ......................... 36
3.4 Broker Connections. .................................. 38

4.1 Work Stealing: Three Hosts Working on a Raytracing Scene. 44
4.2 Work Stealing: Client and Host 1 Steal Work. ............ 45
4.3 Work Stealing: Image Complete. ......................... 46

5.1 Master-Worker ES: Result 4 Arrives. .................. 53
5.2 Master-Worker ES: Result 3 Arrives. .................. 54
5.3 Master-Worker ES: Piece 2 Selected. .................. 55
5.4 Atomic Tasks and Atomic Depth. ....................... 58
5.5 Branch-And-Bound ES: Initial Situation. ............... 62
5.6 Branch-And-Bound ES: Result 4 Arrives. ............... 63
5.7 Branch-And-Bound ES: Result 6 Arrives. ............... 63
5.8 Branch-And-Bound ES: Deferred Detection of Pruned Task 3. 64
5.9 Branch-And-Bound ES: Task 5 Selected. ............... 64
5.10 Large Atomic Task. .................................. 65
5.11 Dynamic Depth Expansion: Large Task Split Up. ....... 66
5.12 Dynamic Depth Expansion: Recursive Splitting. ....... 67
5.13 Dynamic Depth Expansion in Two Locations. .......... 68
5.14 Tree Repair: Host 1 Fails, Host 3 Detects Failure. ..... 71
5.15 Tree Repair: Broker Preempts Host 6. ................ 72
5.16 Tree Repair: Host 6 Replaces Host 1. ................. 72
5.17 Different Failure Scenarios. ......................... 75
6.1 An Instance of the TSP. ................................. 80
6.2 A TSP Search Tree. ................................. 81
6.3 Speedup over T4 for Varying Depth Increments. ......... 85
6.4 Speedup over T20 for TSP on Paderborn Cluster. ........ 87
6.5 Speedup over T64 for TSP on IBM Blue Horizon. .......... 89
6.6 Fault Tolerance on PII 400 Processors. ................... 91
Chapter 1

Introduction

Of late, the emerging field of Global Computing has garnered a lot of interest. Researchers all over the world are trying to harness the Internet’s vast growing computational capacity for ultra-large, coarse-grained parallel applications. A few years ago, three important developments ushered in a new era. One was the announcement of the SETI@home project\textsuperscript{1}, which attracted hundreds of thousands of participants. Since then, similar projects have arisen, including commercial projects like the Fight Against Cancer from Parabon Computation\textsuperscript{2}, folding@home from Stanford\textsuperscript{3}, the anti-cancer drug discovery

\begin{thebibliography}{9}
\bibitem{seti} http://setiathome.ssl.berkeley.edu
\bibitem{parabon} http://www.parabon.com
\bibitem{folding} http://folding.stanford.edu
\end{thebibliography}
project, *Screensaver Lifesaver*, from Oxford in collaboration with United Devices\(^4\), and *Fight AIDS at Home* from Entropia\(^5\). While such applications are of the straightforward master-worker variety, where compute hosts in the system work on self-contained pieces of a large problem, they promote the vision of a world-wide virtual supercomputer \([18, 1]\). However, these projects are all specifically designed to solve a single problem, forcing application programmers to solve the same issues in different contexts each time. Some of these issues are:

- **Correctness** — If the system does not produce correct results, then there is no reason to use it.

- **Heterogeneity** — In order to reach as many participating hosts as possible, the designers of global computing systems must provide code that runs on multiple, heterogeneous computing platforms.

- **Scalable Performance** — If there is no niche where global computing outperforms existing multiprocessor systems, then there is no reason to use it. In order for global computing to outperform existing multiprocessor systems, it must harness a larger set of processors: The architecture

\(^4\)http://www.chem.ox.ac.uk/curecancer.html
\(^5\)http://www.entropia.com
must scale to a higher degree than existing multiprocessor architectures.

- **Fault Tolerance** — An architecture that scales to thousands of hosts must be fault tolerant, particularly when hosts, in addition to failing, may dynamically disassociate from further participation in an ongoing computation.

- **Incentive** — Full use of global computing ultimately implies using a set of computers that is too large for any single person or organization to own or control. Where authority to command is lacking, incentives must be provided [10, 42]. To date, global computing has used fame, fun, or prizes as an incentive (e.g., SETI@home, the Great Internet Mersenne Prime Search⁶, and code-cracking⁷. The Popcorn project [9] has explored computational markets.

The second key development was the advent of *Java*. By providing a portable, secure programming system, *Java* holds the promise of molding the large heterogeneous Internet into a single, homogeneous, multi-user multiprocessor [1]. Some research projects that are designed to exploit this include *Charlotte* [5], *Atlas* [3], *Bayanihan* [37], *Popcorn* [9], *Manta* [40], *Ajents* [27],

⁶http://www.mersenn.org
⁷http://www.distributed.net
and *Globe* [2]. While these efforts automatically overcome the heterogeneity problem, and many provide interesting solutions to task scheduling and fault tolerance, they have not yet achieved a level of scalability and performance to satisfy the demands of application developers.

The third significant development was an emerging vision of *The Grid*, an “integrated, collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations.” [17]. The Globus project [17] is the best known manifestation of that vision. In contrast to Globus, which is open source, Avaki is a proprietary version of that vision, rooted in Legion research [25]. Such systems are not Java-centric, and indeed *must be* language-neutral. The Millennium Project [14] is a “system of systems” project where a key element of the research is how constituent systems interact. The Ninja project [43] is developing an infrastructure for Internet-based *services* that are scalable, fault tolerant, and highly available.

Although most of the research efforts connected to the Grid are more focused on the problem of resource allocation and administration, and less on computational aspects, some application development or deployment systems are *explicitly* based on the Grid, such as AppLeS for parameter sweep computation by Casanova et al. [12], and Condor-G by Frey et al. [20] (where Con-
EveryWare [45] pushed this envelope by presenting an application development toolkit that was demonstrated to "leverage Globus, Legion, Condor, and NetSolve Grid computing infrastructures, the Java language and execution environment, native Windows NT, and native Unix systems in a single, globally distributed application" [45]. In such an environment, the experiment ran on compute resources that were shared by other applications, reaching a scalability level comparable to the results of this dissertation. However, measuring speedup was problematic, and was not done. This is unfortunate for us, since the application, a Ramsey number search, was implemented as a branch-and-bound problem — a programming model that we also support, as described below.

Using the EveryWare toolkit to develop an application that interfaces with this disparate set of components is reputed to require extremely broad expertise. The explicit goal of the GrADS [28] project is to simplify the development and performance tuning of distributed heterogeneous applications destined for the Grid.

This dissertation presents Javelin 3, a Java-based software system for scalable, fault tolerant, adaptively parallel Global Computing. By adaptively parallel, we mean a computational model where the application is oblivious to
how many hosts are participating in the computation — hosts can come and go at any given moment, and need not be pre-allocated by the application. It is a project in the tradition of Charlotte [5] and Cilk/Atlas [3], and in many respects continues where these projects left off. Like the other Java-based efforts above, Javelin 3 is orthogonal to the Grid work: In principle, it can be loosely coupled to the Grid via Grid protocols: Advances in one can be leveraged by the other. The Java CoG Kit[41] facilitates such leveraging activity. Javelin 3 is designed to achieve the following goals:

- Obtain the performance of a massively parallel implementation;
- Provide a simple, easy to use API, allowing designers to focus on a recursive decomposition/composition of the parallelizable part of the computation;
- Reach a scalability level suitable for the very largest applications; and
- Provide transparent fault tolerance, load balancing, and task scheduling.

Summarily, we want the application programmer to get the performance benefits of massive parallelism without the typically attendant costs: adulterating the application logic with interprocessor communication protocols, topology-specific (e.g., hypercube) interprocessor communication, and fault tolerance
schemes. The resulting code should run well on as many processors as are available at any particular execution time, without any change to the program. Indeed, the program should perform well even when the set of processors changes dynamically.

Javelin 3 extends the master-worker computational model to a branch-and-bound model, as described below, which is implemented using a weak form of shared memory that itself is implemented via the pipelined RAM [29] model of cache consistency. This shared memory model is strong enough to support branch-and-bound computation (in particular, bound propagation), but weak enough to be fast.

The task scheduling mechanisms used in Javelin 3 integrate Cilk-like work stealing with an advanced form of eager scheduling, introduced by the Charlotte project. By fine-tuning our eager scheduling methods to support the demands of both programming models, and introducing a dynamic depth decomposition scheme for branch-and-bound eager scheduling, we achieve a degree of scalability that is theoretically unlimited, and practically validated by experiments with over 1,000 host processors.

We also analyze the performance of eager scheduling under a concrete set of assumptions, leading to a simple formula to predict performance degradation
due to host and link failures. The analysis is then validated by experimental results.

The results presented here were achieved on large dedicated parallel machines, for reasons of obtaining clean speedup measurements in a controllable experimental setting. We do not, however, exploit any special features of these machines, in particular, fast communication systems. Our only platform is the standard JDK 1.3, using Java RMI over TCP/IP. We have reported results of previous experiments in heterogeneous LAN environments in [34], and intend to test the Javelin system in a geographically distributed setting in the future.

Model of Computation

The *branch-and-bound method* intelligently enumerates all feasible points of a combinatorial optimization problem. By *intelligent* we mean that not all feasible solutions are examined. Branch-and-bound, in effect, produces a proof that the best solution is found without actually examining all feasible solutions. The method successively partitions the solution space ("branches"), and does not search a subspace ("prunes"), when there is sufficient information to infer that none of the subspace’s solutions are as good as a solution (bound) that already has been found. (See Papadimitriou and Steiglitz [35] for a more complete discussion of branch-and-bound.) Here is a basic, sequential branch-
and-bound algorithm:

```java
activeset = {0}; // "0" is the original problem.
U = infinity;
while (!activeset.empty()) {
    node = activeset.select(); // removes node
    for (int i = 1; i <= node.numChildren; i++)
        if (child[i].lowerBound() < U)
            if (child[i] is complete solution) {
                U = lowerBound[i];
                currentBest = child[i];
            }
    else
        activeset.insert(child[i]);
    // else child is killed implicitly
}
```

The computational model implies the following requirements:

1. Tasks (elements of the active set) are generated during the computation.
2. A host discovering a new bound propagates it to the other hosts.
3. Detecting termination in a distributed implementation requires knowing when all subspaces (child tasks) have been either fully examined or pruned.

The challenge, in sum, is to enable hosts to create tasks, which subsequently can be stolen; to propagate new bounds rapidly to all hosts; and to enable the
eager scheduler to detect tasks that have been completed or pruned\textsuperscript{8}, all with a minimum of communication.

The simpler master-worker model of computation can be seen as a degenerate case of the branch-and-bound model, where:

- all tasks are generated initially (if only in bulk);
- no task is pruned, bound distribution is unnecessary; and
- termination detection is simplified.

In summary, we make the following statement and contributions:

**Thesis Statement**

*It is possible to build an easy to use Global Computing infrastructure that supports the demands of master-worker as well as branch-and-bound applications, provides suitable fault tolerance, and scales up to at least 1000 participating hosts.*

**Contributions**

1. We establish *branch-and-bound* as a feasible new class of application for global computing.

\textsuperscript{8}This is needed not just for termination detection, but also for fault tolerance: determining which tasks may need to be rescheduled.
2. We present two novel *eager scheduling* techniques to support master-worker and branch-and-bound computations.

3. We analyze the performance of eager scheduling under a specific fault model, allowing us to make predictions on performance degradation due to unresponsive hosts.

4. We show experimental results validating our claims on scalability and fault tolerance.

In addition, we provide a simple, convenient API that hides all communication, task scheduling, fault tolerance, and load balancing issues from the application programmer.

**Outline**

The remainder of this dissertation is organized as follows: We begin with an overview of related work in Chapter 2. In Chapter 3, we present the Javelin 3 system architecture, including network structure and code distribution mechanisms. Chapter 4 introduces our approach to scalable computation based on work stealing. In Chapter 5 we describe our two novel approaches to eager scheduling, providing fault tolerance and additional scalability. We also
analyze the performance of our eager scheduling mechanism under specific assumptions. Next, in Chapter 6, we present speedup measurements for large-scale branch-and-bound experiments, as well as results for fault tolerance experiments that validate our previous analysis. Chapter 7 provides an overview of the Javelin 3 API, including code examples of a generic and actual application. Finally, Chapters 8 and 9 conclude this work and present an outlook into the future.
Chapter 2

Related Work

In this chapter we summarize the research efforts that are most closely related to our work. We can broadly categorize related work along the three classes outlined in the introduction:

1. *Large-scale, standalone* global computing applications,

2. *Java-based* network computing projects, and

3. *Grid-centric* research efforts.

The sections of this chapter are organized along these broad classes, with the Java-based class sub-divided into adaptively parallel and fixed parallel programming models. We give an evaluation of the related work in comparison
to Javelin at the end.

2.1 Standalone Applications

We begin our overview by listing three very successful standalone Internet applications, each attracting thousands of participants. None of the generic frameworks below, including our own project, has so far achieved such an enormous scalability. In that sense, these standalone applications set the standards we must aim for.

SETI@home The SETI@home\textsuperscript{1} project (a search for extra-terrestrial intelligence) was a huge success in 1999: shortly after its announcement, over 500,000 users wanted to participate, all based on an incentive of curiosity and fame! These users tried to download the required screensaver, and created a significant scalability problem — the web site was not designed to deal with such a load, and became a bottleneck. During this time, the same part of the computation was passed out and recomputed over and over, and someone labeled SETI@home as the “biggest waste of fossil fuels on the planet.” Despite this initial mishap, the project was a huge

\textsuperscript{1}http://setiathome.ssl.berkeley.edu
success, and its scalability level is so far unmatched.

**GIMPS** The “Great Internet Mersenne Prime Search”\(^2\) is a mathematical project that seeks to find the largest known prime. It has been very successful, finding a number of largest primes since its beginning in 1995. Currently, the 39th know Mersenne Prime has been found: \(2^{46,966,464} - 1\), a number with over 4 million digits! Tens of thousands of computers were involved this search. The incentive for participation is mainly curiosity, but lately, money prizes have been added to attract even higher participation.

**Code Cracking** Distributed.net\(^3\) is one example of the many efforts to crack digital codes through the combined power of many computers. The setup has grown more sophisticated, with a hierarchical structure of server nodes and compute nodes. Distributed.net’s computing power has grown to become equivalent to that of more than 160,000 PII 266MHz computers working 24 hours a day, 7 days a week. Incentives to participate include money prizes, e.g., a $10,000 prize is offered by RSA labs for the current RC5 challenge.

\(^2\)http://www.mersenne.org
\(^3\)http://www.distributed.net
2.2 Programming Models

The following three projects have contributed distinct programming models that have influenced many other projects, including Javelin. Since they are not Java-based, but also adaptively parallel network computing projects, we list them in a separate category.

Cilk [7, 6] Cilk is a C-based runtime system for multithreaded parallel programming on traditional MPPs, although variants have been implemented in other languages and on more loosely coupled workstation clusters (see ATLAS below). It provides a distinct programming model, where a Cilk program consists of a collection of procedures, each of which is broken into a sequence of nonblocking threads. Since threads are non-blocking, they can always run to completion once invoked. Threads can spawn either children or successors, with the latter needed to pick up data dependencies from any children of a predecessor. The programming model is well suited to asynchronous, tree-like computations. Cilk also provides a work-stealing scheduler that enables a processor that runs out of work to steal work from another busy processor, leading to efficient load balancing in the system.
**Linda** [44] Linda is a language for programming parallel applications whose most notable feature is a distributed shared memory called tuple space. A collection of primitives operating on this tuple space allow for interprocess communication and synchronization. Linda implementations are available on a number of different architectures and for a number of different languages. Linda has been used for many applications, including VLSI design, oil exploration, pharmaceutical research, fluid-flow systems design, and financial analysis.

**Piranha** [22, 11] Piranha is one of the first projects aiming at utilizing idle computers in a LAN for parallel applications. It introduces adaptive parallelism, in which the number of processes participating in a computation may vary at runtime. The programming model is a master-worker approach; task communication is handled by a Linda tuple space (see below). The system is fault tolerant by reassigning tasks that have not been completed due to host withdrawal.
2.3 Java-Based Network Computing Projects

2.3.1 Adaptively Parallel Systems

The projects in this section are most closely related to Javelin, and some of them have pioneered important concepts that our system uses. Work stopped early on most of these projects, and none of them reached the scalability level of Javelin 3.

**Charlotte/Knitting Factory** [5, 4] Charlotte supports distributed shared memory, and uses a fork-join model for parallel programming. A distinctive feature of this project is its *eager scheduling* of tasks, where a task may be submitted to several servers, providing fault-tolerance and ensuring timely execution. Charlotte provides fault-tolerance based on the fact that each task is atomic. Changes to the shared memory become visible only after a task completes successfully. This allows a task to be resubmitted to a different server, in case the original server fails.

**ATLAS** [3] ATLAS provides a global computing model based on Java and on the Cilk programming model (see below) that is best suited to tree-like computations. ATLAS ensures scalability using a hierarchy of managers.
The current implementation uses native libraries, which may raise some portability problems. Like Charlotte, ATLAS provides fault-tolerance based on the fact that each task is atomic. Each subtask is computed by a subtree in the hierarchy of servers. Any subtask that does not complete times out and is recomputed from a checkpoint file local to its subtree. Work on ATLAS stopped in a very early stage; scalability was never tested beyond eight processors.

**Popcorn** [9] Popcorn provides a Java API for writing parallel programs for Internet distribution. Applications are decomposed by the programmer into small, self-contained subcomputations, called *computelets*. The application does not specify a destination on which the computelet is to execute. Rather, a centralized entity called “market” brings together buyers and sellers of CPU and determines which seller will run the computelet. Market incentives are supported, e.g. two different types of auction for CPU cycles. User participation as a seller is made extremely easy — just point a Java-enabled browser to the market web site and fill in a user profile to open an account for so-called “Popcoins”, the micro-currency used by Popcorn.
Bayanihan [36, 37] Bayanihan identifies issues and concepts of a Java-based system like Javelin. It classifies the participation in a global computing framework into different levels of volunteer computing, touching on economic concepts (e.g., barter for CPU time without proposing a broker model). The current prototype provides a general framework for executing tasks within a so-called “chassis object” that can either be a Java applet or application. Tasks are dealt out by a centralized server (work manager) and executed by work engines in the client. Communication in Bayanihan is based on the HORB [26] distributed object library. Fault tolerance is provided via eager scheduling, as in Charlotte.

Ninflet [39] Ninflet is a Java-based global computing system with an architecture very similar to Javelin. Clients correspond to Javelin clients, Dispatchers correspond to Javelin brokers, and Servers correspond to Javelin hosts. One major difference is that the system is based on Java applications instead of applets, thus overcoming applet security restrictions and letting the computational units, called Ninflets, establish point-to-point RMI communication. A special Ninflet security model is provided to address security concerns that result from the use of applications. The programming model is a master-slave model in which the client acts
as the master. A special feature of the Ninflet system is the ability to checkpoint ongoing computations in order to provide fault tolerance as well as graceful evacuation of hosts reclaimed by their users.

2.3.2 Traditional Parallel Programming Systems

The following is a list of Java-based systems that share the classic message passing paradigm of parallel programming. As such, they are more distantly related to our work, although they achieve platform independence through Java.

**JPVM** [16] JPVM is a software system based on PVM[38, 21] that is written entirely in Java. Its programming interface is very similar to the C and Fortran interface of PVM, with some enhancements to better suit Java programming. JPVM introduces several new features, e.g. thread safety and multiple communication endpoints per task. The performance of the current prototype is about two orders of magnitude slower than PVM [46].

**IceT** [23, 24] IceT is an environment that tries to combine the portability and interoperability of Java with the superior performance of legacy
code. Here, Java can be used as a control wrapper for existing libraries written in other languages. IceT is similar to PVM in the sense that a daemon process — implemented as a Java application — must run on each participating host; however, it does not require the user to have login access on all hosts. The programming model is explicit message passing. Concepts like process control, migration, and uploading Java bytecode and native libraries to remote hosts are facilitated. A security model is provided to protect participating hosts.

**ParaWeb** [8] ParaWeb provides two separate implementations of a parallel computing infrastructure, each with a different programming model. Their Java Parallel Class Library implementation provides new Java classes that provide a message-passing framework for spawning threads on remote machines and sending and receiving messages. ParaWeb’s Java Parallel Runtime System is implemented by modifying the Java interpreter to provide global shared memory and to allow transparent instantiations of threads on remote machines.
2.4 Grid Computing Projects

Most of the following projects are primarily infrastructure projects, dealing with resource allocation and administration, as well as providing toolkits for application programmers. These efforts are more distantly related to our work, although we envision an eventual integration of Javelin and the Grid. An exception is the EveryWare project, which successfully harnesses multiple Grid resources in one large branch-and-bound experiment. At present, it is our closest competitor in terms of application type and overall scalability.

Condor/Flock of Condors/Condor-G [30, 15, 20] Condor is a distributed batch system aimed at utilizing idle workstations to meet the computing demands of power-hungry users. It does not provide for parallel applications; only single processes can be submitted to the system, where they are queued at a centralized machine manager until a workstation becomes available to execute the job. Condor has user-configurable mechanisms to measure when a machine becomes idle, priority-based scheduling, a remote execution environment that redirects file I/O to the submitting machine, and a mechanism that stops a Condor job when a user reclaims a machine. It also has a sophisticated checkpointing mechanism.
that allows jobs to migrate between machines and make progress even if hosts are frequently withdrawn. Flock of Condors is a further development aimed at interconnecting Condor clusters over wide-area networks. Condor-G is a recent integration effort, where Condor makes use of several Grid services, such as GSIFTP.

Globus [17] Globus is viewed as a networked virtual supercomputer (also known as a metacomputer): an execution environment in which high-speed networks are used to connect supercomputers, databases, scientific instruments, and advanced display devices. The project aims to build a substrate of low-level services — such as communication, resource location and scheduling, authentication, and data access — on which higher-level Grid computing software can be built. A complete toolkit is provided for application and middleware programmers.

Legion [25] Like Globus, Legion strives to provide a single, coherent virtual machine that can accommodate large numbers of hosts. Some of the design goals are scalability, programming ease, fault tolerance, site autonomy, and multi-language support. To achieve these goals and a single, persistent name space, Legion introduces an object model that
wraps around every component of the system.

**EveryWare** [45] EveryWare presents an application development toolkit that was intended to “leverage Globus, Legion, Condor, and NetSolve Grid computing infrastructures, the Java language and execution environment, native Windows NT, and native Unix systems in a single, globally distributed application”[45]. The experiment ran on compute resources that were shared by other applications, reaching a scalability level of over 1,000 host processors. Measuring speedup in such an environment was problematic, and was not done. The application, a Ramsey number search, was implemented as a branch-and-bound problem, making it the closest known competitor for Javelin 3. Resources were adaptively scheduled, and fault tolerance was provided by state replication and redundant task scheduling. Application coding for such a high number of different platforms was tedious, as the authors admit.

### 2.5 Evaluation

Table 2.1 compares the predominant features of the various systems. Here is what the different columns mean:
• Application parallelism can be either fixed or adaptive, where the former defines an application that predetermines its number of processes at startup and keeps this number constant until termination, whereas the latter defines an application that can cope with varying numbers of processes at runtime.

• Programming model describes the way applications must be structured to run on a given system. The most general model is message passing, where the application is given an interface based on send and receive primitives and no other assumptions about application semantics are made. This is also the most traditional way of parallel programming on MPPs, and the most tedious to code. A more restrictive, but easier to use paradigm is master-worker, in which one process is the master that controls all other processes. In such a model the system provides a programming interface for the master and the worker tasks and usually hides communication details like message passing or distributed shared memory (DSM) from the application. Branch-and-bound is a heuristic for search problems, and can be seen as a powerful generalization of master-worker, as stated in the introduction. Work stealing is a concept that was used very successfully by the Cilk project, meaning that idle
processors can steal tasks from their peers to achieve better load balancing. Finally, the Cilk model itself restricts threads to be non-blocking, which means that successor threads must be created to continue procedures that wait for data dependencies from child threads. It is well suited to tree-like computations but has limitations for certain other types of applications. Linda provides another distinct programming model, based on tuplespaces. It is well suited to adaptively parallel computing, but has limitations in terms of scalability: the tuplespace is a central bottleneck which is hard to distribute efficiently.

- **Scalability level** — In this category, MPP means that the system will scale to the size of a massively parallel processor, LAN means that the system can accommodate a cluster the size of a local area network, and WAN — the potentially unlimited case — stands for wide area network, which means that the system can work with hosts outside its immediate domain. Some limitations exist in this category, e.g., if a system has some form of centralized component it will obviously be limited in the total number of hosts it can serve.
- **Fault tolerance** — The most common mechanisms for fault tolerance are: *notification*, which means that the system has some form of resource monitoring and an event model that may notify applications and system components about failures; *checkpointing*, a technique that permits saving the state of a computation at certain intervals so that it can be restarted in case of failures; and *eager scheduling*, where tasks are scheduled to more than one host simultaneously in order to ensure that at least one result is received. *Application* means that the system is leaving all fault tolerance to the application, *none* means the authors do not touch on the subject.

From the table, one can already see that Javelin 3 compares favorably to most other projects. It has the most advanced programming model, allowing for a larger set of potential applications. Its scalability level is higher than all of the Java-based projects, and on par with EveryWare, which is based on The Grid. It compares favorably with EveryWare in terms of speedup measurements (none given for EveryWare) and coding complexity: Javelin is much easier to use by an application programmer than EveryWare. The other Grid computing projects are orthogonal to Javelin. Although they provide some similar features, like the view of a virtual supercomputer and some basic fault
<table>
<thead>
<tr>
<th>Application</th>
<th>Programming Model</th>
<th>Scalability Level</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Javelin 3</td>
<td>adaptive</td>
<td>master-worker, branch-and-bound (work stealing)</td>
<td>WAN/MPP</td>
</tr>
<tr>
<td>Charlotte/Knitting Factory</td>
<td>adaptive</td>
<td>master-worker (DSM)</td>
<td>WAN</td>
</tr>
<tr>
<td>ATLAS</td>
<td>adaptive</td>
<td>Cilk (work stealing)</td>
<td>WAN</td>
</tr>
<tr>
<td>Popcorn</td>
<td>adaptive</td>
<td>master-worker</td>
<td>WAN</td>
</tr>
<tr>
<td>Bayanihan</td>
<td>adaptive</td>
<td>master-worker</td>
<td>LAN</td>
</tr>
<tr>
<td>Ninfile</td>
<td>adaptive</td>
<td>master-worker</td>
<td>LAN</td>
</tr>
<tr>
<td>Piranha</td>
<td>adaptive</td>
<td>master-worker</td>
<td>LAN</td>
</tr>
<tr>
<td>Cilk</td>
<td>adaptive</td>
<td>Cilk (work stealing)</td>
<td>MPP</td>
</tr>
<tr>
<td>ParaWeb</td>
<td>fixed</td>
<td>message passing, threads (DSM)</td>
<td>LAN</td>
</tr>
<tr>
<td>IceT</td>
<td>fixed</td>
<td>message passing</td>
<td>LAN</td>
</tr>
<tr>
<td>JPVM</td>
<td>fixed</td>
<td>message passing</td>
<td>LAN</td>
</tr>
<tr>
<td>Legion</td>
<td>fixed</td>
<td>message passing</td>
<td>WAN</td>
</tr>
<tr>
<td>Globus</td>
<td>fixed</td>
<td>message passing</td>
<td>WAN</td>
</tr>
<tr>
<td>Condor</td>
<td>none</td>
<td>single process (batch)</td>
<td>LAN</td>
</tr>
<tr>
<td>Pack of Condors</td>
<td>none</td>
<td>single process (batch)</td>
<td>WAN</td>
</tr>
<tr>
<td>EveryWare</td>
<td>adaptive</td>
<td>branch-and-bound</td>
<td>WAN</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of Network Computing Projects.
tolerance mechanisms, they are presently focused on infrastructure aspects. More work needs to be done to explore how applications can be conveniently supported, and computation organized in a scalable manner on the Grid.

The standalone application projects were not entered in the table, because they are not aimed at providing a computing framework for multiple applications. However, as already stated, their success in terms of scalability is not yet matched by any generic framework, including Javelin. This is the goal we will strive for in future phases of our project.
Chapter 3

System Architecture

In this chapter we present the Javelin 3 system architecture. We begin with a brief overview of the participating entities and their responsibilities. Next, we explain the Javelin Broker Name Service, which serves as an entry point for any host willing to participate in the system. Then, we give a description of our broker network topology and its host tree preorganization scheme. Finally, we discuss how application code is distributed among hosts so that the computation can proceed.
3.1 Overview

The Javelin 3 system architecture retains the basic structure of its predecessors: Javelin [13], Javelin++ [33], and Javelin 2 [34]. There are three system entities — clients, brokers, and hosts. A client is a process seeking computing resources; a host is a process offering computing resources; a broker is a process that coordinates the allocation of computing resources. Each process is mapped to a single Java Virtual Machine (JVM). Figure 3.1 illustrates the Javelin 3 architecture. Clients register their tasks to be run with a broker; hosts register their intention to run tasks with a broker. The broker assigns tasks to hosts that, then, run the tasks and send results back to the clients. The role of a host or a client is not fixed. A machine may serve as a Javelin host when it is idle (e.g., during night hours), while being a client when its owner wants additional computing resources.

We say that a global computing infrastructure is scalable, when its components have bounded power — bounded computational rate, bounded communication rate, and bounded state\(^\dagger\). In particular, for a system such as Javelin 3 to be scalable, its clients, brokers, and hosts must have bounded power. These bounds imply, for example, that clients, brokers, and hosts can communicate

\(^\dagger\)In this context, bounded stands for bounded by some constant.
with only a fixed number of other components during a fixed interval of time. Thus, at any point in time, there are bounds on the number of connections between hosts, between brokers, between brokers and hosts, and between the client and brokers. Bounded state similarly implies bounds on the number of brokers that a broker can know about at any point in time. We have identified two key problems in building a scalable global computing system:

1. Architectural Scalability — this includes broker discovery, network topology, host allocation, and code distribution, i.e., how does a host find a broker, how does a client find hosts for its computation, and how does the code get distributed efficiently to a potentially very large number of hosts?
2. Computational Scalability — this includes task scheduling and runtime communication between hosts. In other words, how is data exchanged between participating hosts after an application has been successfully started?

In the following, we discuss our approach to solving the architectural scalability problem. Computational scalability will be addressed in Chapter 4. These two chapters are structured according to the different states a Javelin 3 host can be in during its lifetime. The complete state transition diagram is shown in Figure 3.2. There are four states: NoHost, Standby, Ready, and Running. If a host has not joined Javelin 3 it is in state NoHost. The transition to Standby is made by downloading and starting the Javelin 3 daemon, and then finding and registering with a broker. In the next section we describe how brokers are managed, hosts are allocated, and code is shipped so that an application is ready to start, causing a state transition from Standby to Ready. In Chapter 4 we present task scheduling and data exchange mechanisms that allow the host to run the application and therefore transition to Running.

The diagram has two more sets of transitions, a “natural” way back from each state to the previous state when a phase has terminated, and a set of “interrupt” transitions (shown in dashed lines) that lead back to the NoHost
state when a user withdraws the host from the system.

![State Transition Diagram for Javelin 3 Hosts.](image)

Figure 3.2: State Transition Diagram for Javelin 3 Hosts.

### 3.2 Javelin Broker Name Service

When a host (or client) wants to connect to Javelin, it first must find a broker that is willing to serve it. The JavelinBNS system is a scalable, fault tolerant directory service that enables the discovery of a nearby Javelin broker, without any prior knowledge of the broker network structure. It is designed not only to aid hosts who are searching for brokers, but also to aid brokers who are looking for neighboring brokers.

A JavelinBNS system consists of at least two JavelinBNS servers\(^2\). Each server is responsible for managing a list of available brokers, responding to broker lookup requests, and ensuring that the other JavelinBNS nodes contain the same information. The JavelinBNS system thus serves as an information

\(^2\)Information stored by a BNS server is fully replicated.
backbone for the entire Javelin 3 system.

Since the information stored for each broker is relatively small, the service will scale to a very large number of brokers. A small number of BNS servers will therefore be capable of administering thousands of broker entries, so a fully connected network of BNS servers will not be a bottleneck. At regular intervals, information is exchanged by the BNS servers. If a BNS server crashes and subsequently restarts, it can simply reload its tables with the information obtained from its neighbors, thus providing the necessary degree of fault tolerance.

![JavelinBNS Lookup Sequence](image)

Figure 3.3: JavelinBNS Lookup Sequence.

Figure 3.3 shows the steps involved in a broker lookup operation:

1. At startup, a broker registers its address with a known JavelinBNS
server.

2. A host or client willing to participate in Javelin queries the BNS server for a list of \( k \) brokers for some constant \( k \).

3. The BNS server randomly selects up to \( k \) broker addresses from its table and responds.

4. The host sends an RMI ping() call to each broker on the list.

5. The host evaluates the ping results and connects to the most suitable broker, provided the broker is willing to serve it. If not, it can pick the next broker or send another query to the BNS.

Likewise, brokers can themselves use the BNS to fill their address tables with neighboring brokers at startup. The list of known BNS servers is initially loaded from a configuration file, but can be updated by calling the BNS at runtime.

### 3.3 Broker Network & Host Tree

The topology of the broker network is an *unrestricted graph of bounded degree*. Thus, at any time a broker can only communicate with a constant
number of other brokers. This constant may vary among brokers according to their computational power. Similarly, a broker can only handle a constant number of hosts. If that limit is exceeded, hosts must be redirected to other brokers. The bounds on both types of connection give the broker network the potential to scale to arbitrary numbers of participants. At the same time, the degree of connectivity is higher than in a tree-based topology like the one used in the ATLAS project [3]. Figure 3.4 shows the connection setup of a broker.

![Broker Connections Diagram](image)

**Figure 3.4: Broker Connections.**

When a host connects to a broker, the broker enters the host in a logical tree structure. The top-level host in the tree will not receive a parent; instead it will later become a child of the client. This way, the broker maintains a *preorganized tree of hosts* which are set on standby until a client becomes active. When a client connects, or client information is remotely received from
a neighboring broker, the whole tree is activated in a single operation and the client information is passed to the hosts.

Brokers can individually set the branching factors of their trees, and decide how many hosts they can administer. In case of a host failure, the failed node is detected by its children and the broker restructures the tree in a heap-like operation (see Section 5). The additional burden of tree management was previously placed on the client, which could become a bottleneck for large trees. Thus, placing the tree management on the broker enhances scalability and increases performance when a client starts up, since the computation can begin immediately.

3.4 Code Distribution

A client and its local broker do not actively look for hosts to join a computation. Hosts can join at any time, either by contacting the same broker as the client or indirectly through some other broker.

If every host that participates in a computation had to go to the client to download the code this would soon lead to a bottleneck for large numbers of hosts. Therefore, first the local broker and then every other broker that
joins in a computation will act as a cache on behalf of the client. The loading and caching mechanism is implemented as a modification to the standard Java ClassLoader — whenever a loadClass() command fails at a host it is translated to an RMI call to the local broker, which in turn will either deliver the requested class from its cache or make a recursive RMI call to the broker it retrieved the application from. If all calls in this chain fail to deliver the requested class, the client will finally be contacted and deliver the original class file, which will then be cached at all intermediate brokers in the chain. Subsequent requests by other hosts will not reach the client again, thus eliminating another bottleneck in the system. At present, like the standard ClassLoader, the Javelin 3 ClassLoader loads all classes on demand, i.e., only when they are needed.

In the following we describe the sequence of steps from the moment a client application is willing to execute until the moment when a host has received the code to participate in the computation.

1. The client registers with its local broker.

2. The client sends a description of the application to the broker\(^3\). The client may now start up and execute on its own.

\(^3\) currently consisting of the name of the application class and the ID of the client
3. A host joins the system by downloading the Javelin 3 daemon class and starting a JVM that executes the daemon.

4. The host daemon contacts the local broker asking for code to execute.

5. If the local broker has work, it returns the name of the application class and client ID. If not, it contacts its neighboring brokers and asks for code until it either finds an application or all neighbors have denied the request. If this search is successful, the broker also returns the application information to the host.

6. The host daemon executes the above mentioned recursive class loading mechanism to load the application. A new thread is created and the application starts to execute on this host.
Chapter 4

Scalable Computation

In this chapter, we describe how we achieve computational scalability: we first discuss our approach to task scheduling, based on work stealing; then, we explain how we use limited shared memory for asynchronous bound propagation to support the branch-and-bound computational model. Finally, we briefly discuss the underlying non-blocking RMI communication model.

4.1 The Work Stealing Scheduler

The fundamental concept underlying our approach to task scheduling is work stealing, a distributed scheduling scheme made popular by Cilk [7]. Work
stealing is entirely demand driven — when a host runs out of work it requests
work from some host that it knows. One advantage of work stealing is its
natural way of balancing the computational load, as long as the number of
tasks is high relative to the number of hosts — a property well suited for
adaptively parallel systems.

In Javelin 3, tasks get split in a double-ended task queue until a certain
minimum granularity — determined by the application — is reached. Then,
they are processed. The local host computation picks work off one end of the
deque, whereas remote requests get served at the other end. When a host runs
out of local tasks, it selects a neighboring host and requests work from that
host.

Since the hosts are organized as a tree, the selection of the host to steal work
from follows a deterministic algorithm based on the tree structure. Initially,
each host retrieves work from its parent, and computes one task at a time.
When a host finishes all the work in its deque, it attempts to steal work,
first from its children, if any, and, if that fails, from its parent. This strategy
ensures that all the work assigned to the subtree rooted at a host gets done
before that host requests new work from its parent. Work stealing within a
tree of hosts helps each host get a quantity of work that is commensurate with
its capabilities. The client is the root of its tree of hosts.

Figures 4.1 through 4.3 illustrate the work stealing process. In Figure 4.1, three hosts have joined in the computation of a simple raytracing scene. At first, only Host 0 — the client — started work on the whole image. Then, Host 1 joined and stole the left half of the image from Host 0. Next, Host 2 joined and stole the upper right quarter of the image from the client. The figure depicts the situation when all hosts have already completed some atomic pieces of their work.

Figure 4.1: Work Stealing: Three Hosts Working on a Raytracing Scene.

In Figure 4.2, the computation has progressed to the point where first the client was able to steal the upper left quarter of the image back from Host 1,
and then Host 1 in turn stole the upper left eighth of the image back from the client. Host 2 still has work remaining from its initial share of the image. Finally, Figure 4.3 shows the end of the computation with the complete image.

Figure 4.2: Work Stealing: Client and Host 1 Steal Work.

4.2 Shared Memory for Bound Propagation

In the master-worker model of computation, shared memory is not needed. It might appear that we cannot implement shared memory efficiently among geographically distributed processors in a manner that scales; the communication latency is too large. However, for branch-and-bound computation:
Figure 4.3: Work Stealing: Image Complete.

- Only a *small amount* of shared memory is needed, because only one `int` or `double` is needed to represent a solution’s cost.

- A *weak shared memory model* suffices, because if a host’s copy of best cost is stale, correctness is unaffected. Only performance may suffer — we might search a subspace that could be pruned.

It thus suffices to implement the shared memory using a pipelined RAM (aka PRAM) model of cache consistency. This weak cache consistency model can be implemented with scalable performance, even in a geographically distributed setting.
There are several methods to propagate bounds among hosts. We use the following: When a host discovers a solution with a better cost than its cached best cost, it sends this solution to the client. If the client agrees that this indeed is a new best cost solution (it may not be, due to certain race conditions), it updates its cached best cost solution, and “broadcasts” the new best cost to its entire tree of hosts. That is, it propagates the new best cost to its children, who in turn propagate it to their children, and so on, level by level down the host tree. This propagation is handled asynchronously by a separate propagator thread, avoiding the situation where a host blocks until all the hosts in its subtree have acknowledged the new bound.

4.3 Asynchronous Communication Model

Previous versions of Javelin [33, 34] used synchronous RMI for inter-process communication: on communicating with another host, a host would block until the answer was received. Since work stealing requests that could not be satisfied in the local subtree were routed to the parent, and recursively up the tree, there would be many open requests near the end of the computation, when only a few problem pieces were left. In smaller configurations, up to around 100
hosts, this model posed no significant problem — results were generally quite good. However, as we tested larger configurations, the synchronous model became an obstacle to achieving greater scalability: performance degraded, and, when using cluster machines running the Linux OS, the system frequently deadlocked — although in theory, deadlocks should have been resolved at the root of the tree.

Consequently, the communication model was completely redesigned. In the current version, all RMI calls are immediately answered, and state changes recorded, before calls are relayed to other hosts. Thus, no communication links are kept open for long periods, and no hosts are blocked. This was achieved by designing a multi-threaded communication subsystem, in which there are separate threads for work distribution, bound propagation, result forwarding, and the actual computation engine on each host.

These changes have led to much improved scalability in Javelin 3, as our experiments in Section 6 show: the system will now scale to over 1000 processors, and no obvious bottlenecks have been detected.
Chapter 5

Advanced Eager Scheduling

Javelin 3’s advanced *eager scheduler* provides for fault tolerance in the case of host and link failures, as well as load balancing of hosts with varying computational speed. It integrates seamlessly with the work stealing scheduler described in Section 4.1.

Eager scheduling redundantly reschedules a task to an idle processor in case its result has not been reported. It was introduced and made popular by the Charlotte project [5], and also has been used successfully in Bayanihan [37]. It efficiently and relentlessly progresses towards the overall solution in the presence of host and link failures, and varying host processing speeds. In addition, it also provides for balancing of the computational load. The Javelin 3 eager
scheduler is located in the client process, which is at the root of the host tree. Although this may seem like a bottleneck with respect to scalability, it is not, as we shall explain below.

We first describe our eager scheduling strategies for master-worked and branch-and-bound computations. Next, we explain an additional fault tolerance feature of Javelin 3: detecting and correcting failures in the host tree. Finally, we analyze the performance of eager scheduling under certain assumptions and in the presence of a constant host failure rate.

### 5.1 The Master-Worker Eager Scheduler

The master-worker eager scheduler carries over basically unchanged from earlier versions of the system, Javelin++ [33] and Javelin 2 [34]. It employs a fixed-depth tree-based scheme with a circular linked list of undone tasks.

The basic data structure in the master-worker eager scheduler is a heap-like problem tree, maintained in the client process to keep track of the computation status. The tree is pre-generated at the start of the computation, based on two application parameters: the problem’s branching factor, i.e., how many subproblems a single task split will generate; and its atomic depth, i.e., how
man y times a problem can be split before its pieces will be considered atomic. Structurally, the tree is embedded in a circular array in which task sizes are monotonically decreasing. It has a node for every piece of work (task) the problem can possibly generate, and is structured as follows: at its root is the complete, undivided problem itself; its children are the subproblems resulting from a single split of the root problem; and so on, until the atomic pieces of the problem appear at the leaves of the tree. Each node can be in one of three states:

- **done** — meaning the result for the subproblem has been received by the eager scheduler;

- **partly done** — meaning that results have been received by the eager scheduler for some but not all descendants of this subproblem (i.e., some but not all subproblems of this subproblem); and

- **undone** — meaning that no results have been received by the eager scheduler for this subproblem.

Initially, all nodes are in the **undone state**. The processing itself consists of two distinct routines:
1. **Result Processing** — this routine is invoked on all incoming results. Specifically, the eager scheduler marks the subproblem corresponding to the incoming result as *done*, and then recurses up the tree marking ancestors of the subproblem either *done* or *partly done* depending on their current status. At present, the first result for a subproblem is always recorded and passed to the client application’s result handler. Any subsequent results for the same subproblem are simply discarded.\(^1\)

2. **Work Selection** — this routine is invoked only when work stealing from other hosts fails for the client, i.e., the client cannot get work from its children in the host tree. In this case, the eager scheduler selects the next piece of work marked *undone* for rescheduling. Hence, the eager scheduler becomes the *logical parent* of the client in the host tree. Since the tree is embedded in a circular array with piece sizes monotonically decreasing, this piece is guaranteed to be the largest available undone piece. The next time selection is invoked, it will proceed from the current piece to select the next largest undone piece, and so on. At the end of the array, the selection process simply wraps around to the beginning.

\(^1\)Another alternative, to ensure correctness, would be to wait until several results have been received and compare them, and possibly employ some form of quorum-consensus mechanism in case of mismatching results.
and enters another round.

It is important to note that the scheme employed is actually distributed: by selecting the largest available piece of work and reissuing it for processing, the distributed work stealing ensures that this task will be split up and distributed among the participating hosts, just as the pieces were initially. The eager scheduling strategy thus integrates naturally into distributed work stealing.

Figures 5.1 through 5.3 give a simple example of master-worker eager scheduling. In Figure 5.1 we see how the result of the atomic piece with ID 4 arrives at the eager scheduler. The eager scheduler subsequently marks this piece as done and all its ancestors including the root as partly done. In

![Diagram](image)

**Figure 5.1: Master-Worker ES: Result 4 Arrives.**

Figure 5.2 another result arrives for the atomic piece with ID 3. Now the eager
scheduler can also mark the parent of this piece, node 1, as done. Finally, in

![Diagram of a tree with nodes 0, 1, 2, 3, 4, 5, and 6, where nodes 1 and 2 are marked as partly done, and node 0 is marked as done.]

Figure 5.2: Master-Worker ES: Result 3 Arrives.

Figure 5.3 we show how, assuming no further results arrive at the eager scheduler and work stealing has failed, the eager scheduler selects piece number 2 as the largest undone piece of work. This piece will now be reissued for host computation. It subsequently may be split, and parts of it stolen by other hosts.

### 5.2 The Branch-and-Bound Eager Scheduler

Eager scheduling is more challenging for branch-and-bound computation (as compared to master-worker computation). To begin with, the computation
produces two types of result, which must be handled differently:

1. A positive result is a new best cost solution. It is propagated to all hosts as soon as possible, and leads to more efficient pruning of the search tree.

2. A negative result is a solution subspace that has either been fully examined, or pruned from the search tree.

In comparison, a master-worker computation produces only negative results under this terminology. The eager scheduler does not handle positive results in Javelin 3; instead, we use the separate bound propagation mechanism as described in Section 4.2. All negative results are handled by the eager scheduler.
To achieve good performance, unnecessary communication and computation must be avoided. To this end, we do not send the eager scheduler information about pruned nodes. When the scheduler must know about this information (when no work is left to steal), it may have a better minimum cost bound, enabling it to prune at a higher level in the problem tree. This reduces its overall workload. We also aggregate the communication of negative results, to improve the scheduler’s computation/communication ratio, by collecting results for a given time interval (e.g., 5 seconds), and sending them up the tree to the parent periodically. Too much communication to the eager scheduler would make it a bottleneck, and thus negate the benefits of using multiple hosts.

In a branch-and-bound computation, the size of the feasible solution space is typically exponential in the size of the input. In principle, the algorithm may need to examine all of these exponentially many feasible solutions to find the minimum cost solution. In practice, a partial solution, \( p \), is “pruned” when the lower bound on the cost of any feasible solution that is an extension of \( p \) must be more costly than the currently known minimum cost solution. The algorithm nonetheless must gather sufficient information to detect that the minimum cost solution has indeed been found. This implies that pruned
nodes and sub-optimal solutions must be detected by the eager scheduler. If a separate communication is required to detect each such event, the overall quantity of communication would nullify the benefits of parallelism. We cope with this communication overload by aggregating portions of the search space into atomic tasks, and similarly aggregating negative results into one communication per atomic task. This lets the eager scheduler know that this part of the problem tree has been searched, and hence need not be rescheduled. The number of negative communications consequently is equal to or less than the number of atomic tasks. In practice, it is much less than the number of atomic tasks; many are pruned. We can adjust the computation/communication ratio by adjusting the size of atomic tasks, in order to decrease the overall run time. Performance is quite sensitive to atomic task size, so finding good size values is important.

Figure 5.4 shows an example of the search tree division into atomic and non-atomic tasks. Here, the atomic depth parameter is set to 2, which means all tasks that are 2 nodes below the root node are considered atomic and will be processed by hosts with no further subdivision. Many of these tasks may in fact be pruned away, due to an existing better minimum cost bound. We have highlighted two portions of the tree that show the extent of local computation
of an atomic task. The eager scheduler itself never processes any portions of
the search tree below the atomic depth level, although it will analyze the top
portion of the tree based on negative results and its own minimum cost bound.

Figure 5.4: Atomic Tasks and Atomic Depth.

5.2.1 Deferred Detection of Pruned Nodes

The branch-and-bound eager scheduler employs *lazy* or *deferred* detection
of pruned nodes, which takes place only after the initial round of work stealing
has ended, and the first host finds itself out of work and sends a work request
to the eager scheduler. At this point the eager scheduler infers what nodes
representing partial solutions have been pruned, and which atomic tasks need to be rescheduled.

The advantage of deferred pruning is that, when pruned nodes need to be identified, usually near the end of a computation, a tighter minimum cost bound is known. This allows the eager scheduler to prune tasks higher in the problem tree, thus resulting in less total work for the eager scheduler. A disadvantage is that only atomic tasks can be eagerly scheduled, because hosts do not report pruned nodes to the eager scheduler. If a task were rescheduled whose subtasks are then pruned, the eager scheduler would never receive this information, thus would eagerly schedule the task forever.

5.2.2 Method

The basic data structure required is a problem tree, which the eager scheduler, located in the client process, maintains to keep track of the computation status. Each atomic task is a leaf in this problem tree. The root of the problem tree represents the complete branch-and-bound computation. Its children are the subproblems resulting from branching — a single split of the root problem. This branching (splitting) continues, as we proceed down the problem tree: We subdivide it into smaller and smaller search spaces. A parameter,
the atomic depth, determines at what level splitting stops. At that point, a
host will search the space for a solution that is less than the current minimum
cost solution.

As with the master-worker model of computation, each node (task) in
the problem tree can be in one of 3 states: done, meaning the results for
the subproblem have been received by the eager scheduler; partially done,
meaning that results have been received by the eager scheduler for some but
not all descendants of this subproblem (i.e., some but not all subproblems of
this subproblem); and undone, meaning that no results have been received by
the eager scheduler for this subproblem.

In addition to the tree structure, undone tasks are put in a circular linked
list. Tasks are eagerly scheduled from this circular list until it becomes empty:
there are no undone tasks. This indicates completion of the computation; the
eager scheduler then propagates a termination signal down the host tree. The
processing itself consists of two distinct routines: result processing and task
selection, given below.

```java
public void processResult(Task t) {
    insert t into ProblemTree;
    mark t done;
    mark its ancestors in ProblemTree as either
    partially done or done, as appropriate;
    maintain undone task list;
}```
public Task selectTask()
{
    // esTask refers to last eagerly scheduled node
    while ( (esTask = esTask.next() ) != null ){
        while ( esTask != null && !esTask.isAtomic() ){
            generate esTask’s feasible children & their costs;
            insert children into ProblemTree;
            maintain undone task list;
            esTask = selectTask(one of these children);
        }
        if ( esTask.isAtomic() ){
            if ( esTask.hasBeenRescheduled() ) {
                // dynamic depth increment
                esTask.incrementAtomicDepth();
                continue;
            }
            return esTask;       // found atomic node to process
        }
        // else no feasible atomic node on this path of ProblemTree
    }
    done = true;    // set terminate signal
    return null;
}

Figures 5.5 through 5.9 give an example of the branch-and-bound eager scheduler. One significant difference to master-worker eager scheduling is that here, the tree is not pre-generated — doing so would be prohibitive for memory reasons (branch-and-bound search tree can be huge, and we only need to process a small portion at a time). Thus, Figure 5.5 shows the initial situation in which only the root node is inserted into the tree. In Figure 5.6 we see how
the result of the atomic piece with ID 4 arrives at the eager scheduler. The eager scheduler subsequently marks it as done, and all its ancestors including the root as partly done. In Figure 5.7 another result arrives for the atomic piece with ID 6.

Figure 5.8 shows how, assuming no further results arrive at the eager scheduler and work stealing has failed, the eager scheduler analyzes the current situation: based on its current minimum cost bound, it infers that the node with ID 3 was pruned from the tree, and subsequently marks node 1 as done. Finally, in Figure 5.9 we show how the eager scheduler selects node/task 5 as the next undone piece of work. This piece will now be reissued for host computation.
Figure 5.6: Branch-And-Bound ES: Result 4 Arrives.

Figure 5.7: Branch-And-Bound ES: Result 6 Arrives.
Figure 5.8: Branch-And-Bound ES: Deferred Detection of Pruned Task 3.

Figure 5.9: Branch-And-Bound ES: Task 5 Selected.
5.2.3 Dynamic Depth Expansion

Our experience has shown that, due to the inherent irregularity of many branch-and-bound problems (e.g., the TSP), computation times for tasks with a fixed atomic depth parameter vary greatly; in fact, some pieces compute in milliseconds, as they are quickly pruned; others may take more than half the time of the complete computation. Figure 5.10 shows a situation in which a single large atomic task forms the bulk of the computation.

![Diagram showing large atomic task](image)

Figure 5.10: Large Atomic Task.

We therefore improved upon our fixed-depth eager scheduling scheme de-
scribed in [34] by adding a *dynamic depth* component, which increases the atomic depth of eagerly scheduled pieces on each new round. Thus, larger tasks can be sub-divided into tasks representing smaller subproblems and distributed to several hosts. This dynamic depth expansion has greatly improved the performance and scalability of our system (see Section 6). Figure 5.11 shows an example of this mechanism: a large atomic task is sub-divided by incrementing the atomic depth. The process can be repeated, as shown in

![Single depth increment](image)

**Figure 5.11: Dynamic Depth Expansion: Large Task Split Up.**

Fig. 5.12, if necessary. Finally, Figure 5.13 shows a situation where the atomic depth has been locally incremented in two places (only the portion of the
double depth increment

Figure 5.12: Dynamic Depth Expansion: Recursive Splitting.
problem tree visible to the eager scheduler is depicted).

![Diagram](image)

Figure 5.13: Dynamic Depth Expansion in Two Locations.

### 5.2.4 Alternative: Eager Detection of Pruned Nodes

In an alternative scheme, with which we experimented earlier, a host communicates to the eager scheduler only when it has found a less costly solution than its cached minimum cost solution. Specifically, it communicates only the least cost solutions of atomic tasks that contain a solution that is less than its cached minimum cost solution. However, along with that communication, it conveys other atomic tasks that it has done, including nodes that it has pruned. Thus, the number of communications necessary to provide sufficient information to detect termination is less than the number of nodes in the fringe of the tree, where the fringe is the set of nodes that represent either atomic
tasks or pruned partial solutions. Large tasks (i.e., tasks that are high in the problem tree) can be rescheduled. This results in hosts splitting these tasks, allowing the rescheduling and computation of their subtasks to be distributed (via work stealing) among the hosts.

This alternative scheme may communicate more pruned nodes than ultimately is necessary because it communicates them sooner, when the minimum cost bound may not be so tight. However, the eager detection scheme has the advantage that non-atomic tasks may be rescheduled, which enables the rescheduling and computation of subtasks to be distributed among the hosts. On the other hand, a host does not report negative results and pruned subtrees until a new minimum cost solution is found. In the meantime, work that it has completed might be needlessly rescheduled. It thus is not clear to us which eager scheduling method would perform better overall for a given branch-and-bound computation, much less for most branch-and-bound computations. In our experiments, both methods showed no significant difference in performance.
5.3 Repairing the Host Tree

If a host located at a leaf position in the tree fails (or retreats from the system), it does not affect other hosts very much, since the failed host, when detected, will just be taken off the detecting host’s address list. In case of a non-leaf host failure, however, a host failure has more consequences. Depending on its position in the host tree, a failed host blocks communication among its children, and a portion of the undone computation residing at the failed host may never be assigned to any host in its subtree. Eager scheduling guarantees that the work will be done eventually by the hosts that remain accessible to the client, but it clearly would be desirable to fix a broken tree structure as fast as possible, especially if the failed host is the root of a large subtree of hosts.

Javelin 3 provides a feature that automatically fixes a tree as soon as a host failure is detected. As a precondition we make the assumption that the broker is a stable participant, since it maintains the tree manager.

The tree repair scheme works as follows: When a host is assigned a position in the tree, it is given information on how to contact its parent. If a host later detects that its parent is dead, it immediately notifies the broker of this
condition. If the empty position has already been reported and filled, the tree manager traverses the tree representation, and returns the new parent to the host. However, if the host is the first to report the failure, the tree manager reheap the tree. First, it notifies the last node in the tree, which is to be moved to fix the gap. Figures 5.14 through 5.16 illustrate the situation where node 1 has failed and is replaced by node 6, which is moved to its new position.

![Tree Repair Diagram]

Figure 5.14: Tree Repair: Host 1 Fails, Host 3 Detects Failure.

At present, the tree repair scheme can only cope with host failures, i.e., all hosts that detect a failure must agree on that observation. A single host is not able to distinguish between host and link (communication) failures; the result is the same from the point of view of the host. In case of a link failure between a host and only one of its children, the present scheme reports a failure to
Figure 5.15: Tree Repair: Broker Preempts Host 6.

Figure 5.16: Tree Repair: Host 6 Replaces Host 1.
the broker even when sibling hosts can still communicate with the “failed” parent host. Clearly, what is needed is a form of quorum consensus algorithm. Therefore, our basic scheme needs to respond in a more sophisticated way to link failures. This is a topic of future research in Javelin 3.

5.4 Eager Scheduling Analysis

5.4.1 Leaf Node Failures

Let $T_n$ denote the running time of an experiment with $n$ hosts with identical compute power. If a leaf node fails in the host tree, eager scheduling guarantees that all lost work, i.e., all the pieces that the failed host was currently working on or that had not been reported back to the eager scheduler yet, will eventually be rescheduled. Thus, an upper bound on the running time with a single failure would trivially be $T_{n-1}$. Under the same reasoning, if there are $k$ leaf node failures, the expected running time would be at least as fast as $T_{n-k}$. Since we assume the client to be a stable participant, in the worst case scenario the expected running time degenerates to $T_1$, if all other hosts fail. If we make a few more assumptions, we obtain a much tighter bound on the expected running time:
1. The time to calculate an individual, atomic piece of work $\Delta t$ is negligible in comparison to the total compute time of the problem $T$, or $\Delta t \ll T$.

2. The average communication latency $l$ between hosts is negligible in comparison to $\Delta t$, or $l \ll \Delta t$.

3. The total number of failures $k$ is small compared to the number of initial hosts, $n$.

In the absence of failures, with $n$ identical hosts participating in the computation, each host contributes approximately $1/n$ to the computation. Under the above assumptions, if a failure occurs at time $t$ somewhere between the beginning of the computation, $t_0$, and the end, $t_1$ (with $T = t_1 - t_0$), then the contribution of the failed host is approximated by $\frac{t - t_0}{T}$. A special case occurs if the failure happens exactly halfway through the computation, as shown in Figure 5.17(a). Here, the contribution of the failed host is $1/2n$. This means that, if 2 hosts fail mid-way through the computation, the effect is as if we had one fewer host throughout the whole computation. Let $E_T(i)$ denote the expected running time for $i$ failures. Hence, $E_T(2) = T_{n-1}$, or, for any $k$ halfway failures, the expected running time

$$E_T(k) = T_{n-k/2}.$$
We now show that the above result holds even under failures that don’t occur exactly halfway through the computation: For instance, if two failures occur at times that are symmetric to the halfway point, the effect is as if both failures had occurred mid-way. Figure 5.17(b) illustrates the example.

![Diagram](image)

Figure 5.17: Different Failure Scenarios.

Consequently, even under the assumption of a constant failure rate, as depicted in Figure 5.17(c), the effects of failures will combine pairwise, and the above formula holds — as long as failures are approximately symmetric to the halfway point. Obviously, if the failure rate is too high, the boundary assumptions are violated, and performance eventually degenerates.
5.4.2 Non-Leaf Node Failures

We now look at what happens if a non-leaf node, i.e., a host situated higher up in the tree, fails. Depending on its position in the host tree, a failed host blocks communication among its children, and a portion of the undone computation residing at the failed host may never be assigned to any host in its subtree. Eager scheduling guarantees that the work is done eventually by the hosts that remain accessible to the client.

Let $h(i)$ denote the height of node $i$ in the host tree, with the height of a leaf node being 0. Let $b$ denote the branching factor of the tree. Then, a failed node cuts off $k = \sum_{j=0}^{h(i)} b^j$ hosts in the tree. Since the cut off hosts are treated like any other failed hosts by the system, we simply plug this number into the above formula to predict the overall running time, as before.

5.4.3 Tree Repair

Under the tree repair scheme described above, the consequences of a host failure higher up the tree are again reduced to a leaf node failure, since the time involved to fix the tree is on the order of several hundred milliseconds — which is very short compared to the computation time of an atomic piece
in a typical computation, satisfying the assumptions above. Thus, the above
analysis for leaf node failures holds. Our experiments, described in Section 6.2,
verify that this is indeed the case in practice.
Chapter 6

Experimental Results

In this chapter, we present experimental results for Javelin 3. We begin with a series of experiments designed to demonstrate the scalability of our system, followed by experiments to validate the theoretical results of the preceding fault tolerance analysis. For these tests, we had access to three dedicated parallel machines:

1. A 96-processor Beowulf cluster at our own Computer Science Department at UC Santa Barbara. This machine has six 500 MHz Pentium III quad-processor nodes, and 36 400 MHz Pentium II dual processors, each with 512 MB or 1 GB of memory, running Red Hat Linux 6.2.
2. A 192-processor Beowulf cluster at University of Paderborn, Germany. This machine has 96 850 MHz Pentium III dual-processor nodes, each with 512 MB memory, running Red Hat Linux 7.1.

3. An 1152-processor IBM Blue Horizon at NPARC, San Diego. The machine has 144 SP Power3 8-processor nodes, clocked at 375 MHz, with 4 GB of memory per node. The OS is AIX.

We tested our system on these dedicated architectures in order to obtain clean speedup curves in the classical sense, and to work in a controlled test environment. Previously, we published results obtained on a LAN [33]; the system should also perform well in a geographically distributed WAN.

6.1 Scalability Experiments

6.1.1 TSP Test Application

For the scalability experiments, we chose to run a classical branch-and-bound application, the Traveling Salesman Problem (TSP). In brief, the TSP can be stated as follows:

\footnote{According to the latest Top 500 list, this is currently the 18th most powerful machine in the world.}
Given a weighted, directed graph $G = (V,E)$, find a *minimum weight tour* such that each $v \in V$ is visited exactly once.

Figure 6.1 shows a simple instance of the TSP. Here, the graph is undirected, which can be viewed as a special case of a directed graph with both corresponding directed edges having identical weight. The minimum tour for this instance is highlighted in the figure.

Figure 6.1: An Instance of the TSP.

Figure 6.2 shows the corresponding complete search tree for this simple instance. From this it becomes clear that even for very small instances of the problem, search trees can be huge. In fact, it is well known that the TSP falls into the category of NP-complete problems, for which all currently known algorithms leading to a general solution have at least exponential complexity.

Our TSP application uses a relatively simple, depth-first algorithm for local computation. A breadth-first component is introduced by running the problem
in parallel. A code example is given in Section 7.3. The current version was optimized (compared to the unoptimized results published in the year 2000 [34]) for local computation through the addition of a Kruskal Minimum Spanning Tree lower bound estimate. While this led to dramatic improvements in local computation speed, and let us process much larger graphs (35–37 nodes, instead of 22–24 nodes previously), it made the overall task of achieving good speedup and scalability harder, since communication latencies have not improved.
6.1.2 Finding Suitable Input Graphs

For our largest experiments, the test graphs were complete, undirected, weighted graphs of 35–37 nodes, with randomly generated integer edge weights \( w, 0 \leq w < 1000 \). These graphs were complex enough to justify parallel computing, but small enough to enable us to run tests in a reasonable amount of time. By using complete graphs as input, we made sure that the graphs were as dense as possible, making this type of graph the hardest to process for a given size. Also, we are not exploiting any special cases like for instance in the Euclidean TSP, where all edge weights fulfill the triangle inequalities, and which can therefore be bounded from above by two times the weight of its minimum spanning tree.

The procedure of finding input graphs itself proved to be tedious: we generated some 45 candidate graphs over time, and tested them on a small set of hosts — initially 20, later 64 — to get an idea of the problem size. Many of the candidate graphs proved either too simple (running time too short) or too hard (running time too long). We considered graphs that ran in approximately 10–20 hours on 64 processors of the Paderborn Linux cluster. Our aim was to find graphs that would run for approximately 2-4 hours on 1000 processors on NPACT’s IBM Blue Horizon. Table 6.1 shows some characteristic data for
the graphs that we used in our large scale experiments, measured on IBM Blue Horizon. In addition to the 64-processor running times, the table shows the average time per atomic task and the maximum time per atomic task. Minimum times are on the order of several milliseconds; thus, the variance in atomic task sizes is huge. Note how, for graph 37e, the average time per task decreases by over 50% when the dynamic depth increment is changed from 1 to 2 — this is indeed the desired effect of splitting up large pieces.

<table>
<thead>
<tr>
<th>graph</th>
<th>depth incr.</th>
<th>T 64</th>
<th>avg/task</th>
<th>max/task</th>
</tr>
</thead>
<tbody>
<tr>
<td>35k</td>
<td>1</td>
<td>13.31 h</td>
<td>222 s</td>
<td>8.27 h</td>
</tr>
<tr>
<td>36c</td>
<td>2</td>
<td>21.87 h</td>
<td>177 s</td>
<td>9.18 h</td>
</tr>
<tr>
<td>37e</td>
<td>1</td>
<td>13.88 h</td>
<td>430 s</td>
<td>6.93 h</td>
</tr>
<tr>
<td>37e</td>
<td>2</td>
<td>12.39 h</td>
<td>200 s</td>
<td>6.57 h</td>
</tr>
</tbody>
</table>

Table 6.1: Input Graph Parameters for TSP on IBM Blue Horizon.

### 6.1.3 The Effect of Dynamic Depth Expansion

To demonstrate the effect of our dynamic depth expansion, described in Section 5.2.3, we varied the depth increment parameter from 0 through 5 for a given test graph. Figure 6.3 shows a set of speedup curves for a smaller, 32-node graph, run on the Paderborn Linux cluster. Although these are speedup curves in the classical sense, the base case is not measured on a single processor.
Instead, we show speedup over T4, the time it took 4 processors to compute the test graph. The reasons for this are twofold:

1. The time to complete a run with 4 processors is approximately 3 hours. The time to run on a single processor is therefore prohibitively long, and since speedup in that region of the curve is close to optimal, does not add anything qualitatively.

2. In taking a 4-processor experiment as our base case, we eliminate some of the potential for observing superlinear speedup by adding a breadth-first component to our local depth-first algorithm.

The results show that speedup was mediocre with 0 increment (\(dinc0\) in the diagram), i.e., with a static task size. This is due to a single large piece in this graph that imbalances the load and lengthens the critical path. With an increment of 1 per round, the result is much improved. Increments of 2 and 3 yield even better speedup, although the improvement is not so dramatic. The best results were obtained with an increment of 4 for this graph, at least until the saturation point was reached at 64 processors and the overall running time became too short. An increment of 5 then performed much worse, about the same as 1. This is due to the very large number of tasks per round generated by
such high increments, placing an additional burden on the client process and the communication system. Other experiments with different graphs generally confirmed the following observations:

- For each graph, there is an individual optimum depth increment, usually around 2 or 3.
- The more imbalance a graph shows with 0 increment, the more likely it will be that a higher increment yields a better result.
• Dynamic depth expansion is most useful in the center of the curve, when overall running times are still long enough to split up large pieces. For the smaller host configurations, it is often not needed to achieve good results; in the very largest configurations, the short total running time means that some remaining large pieces will still prolong the critical path.

In summary, this feature has greatly improved the system’s performance and let us achieve good speedup for graphs that were previously deemed unsuitable.

6.1.4 Large Scale Experiments

We present two sets of recent results for Javelin 3. Figure 6.4 shows two speedup curves, run on a 192-processor Beowulf cluster at University of Paderborn, Germany. We show speedup over T20, the time it took 20 processors to compute the test graph. For these graphs, the depth increment was set to 1. The results show that speedup was very good for both tested graphs, with no obvious sign of tapering off at 160 or 180 processors. With the addition of the variable depth increment, we were able to achieve similarly good speedup for almost every graph we tested on this machine, as long as it was large enough
to fully utilize all 192 processors.

![Graph showing speedup over T20 for TSP on Paderborn Cluster.](image)

**Figure 6.4:** Speedup over T20 for TSP on Paderborn Cluster.

The next set of results, shown in Figure 6.5, were run on the IBM Blue Horizon. In direct comparison, its processors are approximately 1.4 times slower than the Paderborn processors. Hence, 192 processors in Paderborn are approximately equivalent to about 256 processors in San Diego. The input graphs have approximately the same size as the larger of the two graphs tested in Paderborn. Here, we chose a base case of 64 processors, for the same reasons as stated above. The results show the same good speedup as the Paderborn results, up to around 500 processors. Again, we can show the
benefit of dynamic depth expansion: for graph 37ε, with an increment of 1, speedup dropped off strongly at 500 processors. With an increment of 2, the result was better in every configuration. Interestingly, our best result, for graph 35k, was achieved with depth increment 1 — the graph showed no further improvement for higher increments, which means that it was evenly balanced to begin with. The largest measurements, on up to 1024 processors, show some dropoff in speedup and efficiency. We attribute this to the overall increase in communication-to-computation ratio: the total time to compute the problem with 1024 hosts was between 40 minutes and 2 hours, depending on the graph. Our experience shows that communication overhead becomes significant when overall running time drops below one hour. Also, any large pieces not yet split up by dynamic depth expansion lengthen the critical path. At this point, we are convinced that the system is not at fault — if we increase the size of the test graph, we should see better speedup. This, however, was not possible with the current allocation of 20,000 units on Blue Horizon. Instead, we would require around 80,000 units for such large tests.

Overall, these results are much improved, compared to earlier versions of Javelin: Due to algorithmic improvements, faster processors, and the advent of a faster JVM in the JDK 1.3 release, we are able to test graphs whose
search trees are several orders of magnitude larger, even though communication latency remains the same. Scalability is significantly higher, too: we were able to fully utilize up to 1024 processors on the IBM Blue Horizon machine, without reaching an obvious scalability limit of our software.

## 6.2 Fault Tolerance Experiments

Next, we present some experiments designed to validate our eager scheduling analysis. In these tests, we chose to run a simple raytracing application:
as a classical example of a master-worker computation, the variance between the running times of the atomic tasks is relatively low, whereas in a branch-and-bound example like the TSP, running times vary significantly. Too much variance in the time to calculate individual pieces violates the first of the assumptions made in Section 5.4, as the time to calculate an individual, atomic piece of work $\Delta t$ will not be negligible in comparison to the total compute time of the problem $T$.

In these tests, we rendered a scene of 5223 objects on up to 10 hosts, using the 400 MHz dual processor nodes of our local Linux cluster. Figure 6.6 shows a comparison of running times with and without induced failures. The right columns show failure-free runs of 9, 8, 7, and 6 hosts. The left columns show results of tests in which we killed a number of hosts at times symmetric to the halfway point of the computation. In the leftmost experiment, we started with 10 hosts, and killed 2 during the computation. According to our theory, $T_{10}(2)$ should equal $T_8$; the measured result was indeed within 3% of this prediction. The next result, $T_{10}(4)$, was within 4% of the predicted value, $T_8$. $T_{10}(6)$ was approximately 6% slower than its predicted value, $T_7$. Finally, $T_{10}(8)$, a test in which we killed 8 out of 10 hosts during the computation, was about 7.5% off the target, $T_6$. 

90
The increasing gap between the times with and without failures can be explained by the non-negligible loss of work in this scenario: The average running times of the atomic tasks were about 27 secs, and maximum times were as large as 57 secs. The more hosts are killed, the more work needs to be rescheduled and redistributed; a small overhead is to be expected. Overall, we consider these results a strong confirmation of our theoretical analysis.

![Figure 6.6: Fault Tolerance on PII 400 Processors.](image)
Chapter 7

The Javelin 3 API

In this chapter we illustrate our system from an application programmer’s point of view. We first present the classes and interfaces needed by the programmer to create a Javelin 3 application. We then demonstrate how the API is used in a generic client and host example that shows how easy it is to create Javelin 3 applications. Finally, we give a concrete example of our TSP application. Most of the API looks very similar to the previous Javelin++ API described in [32, 33], although a few changes were necessary to support branch-and-bound applications.

A Javelin 3 application consists of one client and many hosts. The client is responsible for initiating the computation, managing the problem, and col-
lecting the results. It may or may not do part of the actual computation. The hosts help the client compute the problem. The client code executes on a single machine, while the host code is distributed throughout the Javelin 3 network and executed on many different machines.

All of the Javelin 3 system classes are contained in three packages: javelin, javelin.util, and javelin.id. The first package contains all of the core Javelin 3 classes (all interface classes needed by the application, as well as the executable classes for the broker and host), the second one contains internal data structures and other helper classes, and the final package consists of internal IDs and system parameters. Since the latter two packages are not normally visible to the application, we shall not further describe them here. We follow the convention that all classes and interfaces beginning with the letter “J” are implemented in Javelin 3 and can be directly used by the application, whereas interfaces not beginning with “J” must be implemented by the application in order to work with the system.

The application programmer must provide code for both the client and the host — which may actually be joined together in a single source file, as our TSP example below shows — plus the implementation of three interfaces needed by the system.
7.1 The javelin Package

This package contains all the core classes needed by clients, hosts, and brokers. The programmer writing an application for Javelin 3 first needs to get acquainted with the JavelinClient class.

```java
public class JavelinClient {
    public JavelinClient(String className,
                          String broker,
                          Serializable inputData);

    public JavelinClient(String className,
                          String broker);

    public void begin();
    public void terminate();
}
```

All Javelin 3 clients must create an instance of JavelinClient. The main constructor of JavelinClient takes the top-level class name (used to load the host classes), the broker’s host name, and any input data that hosts may need as arguments. A second constructor is provided for applications that don’t need to pass any input data to their hosts.

Once the client is ready to start the computation, the client invokes the begin() method. The begin() method causes the client to register with a broker, which in turn allows for the broker network to assign hosts to the
client’s computation. The method will also send the client’s RMI handle to
the broker, which will be cached on the broker and passed on to a host upon
registration. The `terminate()` method unregisters the client, allowing the
broker network to clean up and stop assigning hosts to that client. It should
be called after the computation is done to detach gracefully from the broker.

To manage the computation, the Javelin 3 runtime system instantiates a
data manager object in the client and host processes. The data manager is
the heart of the Javelin 3 system: it coordinates the work stealing process (as
described in Chapter 4, hides all inter-process RMI communication from the
application, manages the double-ended task queue, and interacts with the ea-
erg scheduler on the client process. The data manager also sets up threads for
result reporting and bound propagation. In summary, it is responsible for pro-
viding scalable computation and fault tolerance. Currently, Javelin 3 provides
only one specific data manager, called `JavelinDataManager`. It implements
the `JDataManager` interface shown below.

```java
public interface JDataManager {
    public void addWork(Splittable work);
    public Splittable getWork();
    public void returnMWResult(Serializable result);
    public void returnMWResult(Serializable result, long time);
}
```
The main methods are *addWork()*, *getWork()* and the various versions of *returnResult()*. In our model of computation, the *addWork()* method is typically only executed once by the client to initialize the computation, by entering the undivided (root) object of the computation. The *getWork()* method is used by a host to obtain an *atomic* piece of the computation, which has been internally split by the data manager. In case the local computation produces a usable result, the host passes that result back to the data manager using the appropriate *returnResult()* method — *returnBNBResult* for a branch-and-bound computation, or *returnMWResult* for a master-worker computation. The data manager will then transparently communicate the result to the client, by passing it up the host tree.
Two methods were added to support branch-and-bound computations: The `setBNB()` method allows the client application to tell the data manager that it will run as a branch-and-bound computation; the default computation type is master-worker. The `propagateValue()` method is provided to support the TSP application’s bound propagation. It passes on a newly discovered local optimum bound value to the data manager’s propagation thread, which in turn sends the new bound asynchronously to the client and all its children. In this way, bounds are spread recursively through the host tree, supplying the mechanism to implement the weak consistency model described in 4.

The programmer must also tell the data manager how to notify the application whenever a new result arrives and when all the work is complete. This is done by the methods `setResultListener()` and `setDoneListener()`. The `ResultListener` interface is needed by the client, so that the data manager can notify the application’s own result handling routine. The `DoneListener` interface provides a way for the application to specify its own cleanup procedure upon termination. The programmer must implement the two interfaces below so that the respective methods can be called by the system.

```java
public interface ResultListener {
    public void handleResult(Serializable result);
}
```
public interface DoneListener
{
    public void workDone();
}

We now describe how the client conveys its work definition to the data manager. For this, the programmer defines a class, representing the type of work to be done, that implements the Splittable interface or one of its subtypes, shown below. The data manager uses the methods declared in Splittable to recursively divide the work in the deque when the local host computation requests an atomic task. There are currently two interfaces derived from Splittable, named MWSplittable and BNBSplittable, which provide for additional methods needed by the respective type of computation.

public interface Splittable extends Serializable
{
    public boolean canSplit();
    public Splittable [] split();
}

public interface MWSplittable extends Splittable
{
    public int getSplitSize();
    public int getObjectSize();
}

public interface BNBSplittable extends Splittable
{
    public int getMaxDepth();
}
```java
public int getAtomicDepth();
public void incAtomicDepth(int depthIncrement);
}
```

The `split()` method splits the work represented by a particular object into suitable sub-objects. The results of the split are returned in an array of Splittable objects. For example, assume we have a class that implements the `Splittable` interface and represents an image. If we were to invoke the `split()` method on an instance representing an $n \times n$ image, the returned array could contain two new instances each representing an $\frac{n}{2} \times n$ image. The `canSplit()` method determines if a split is possible and is always invoked prior to the `split()` method. If `canSplit()` returns `false`, the `split()` method will not be called.

The `getSplitSize()` method in `MWSplittable` is used to determine the size of a single split in the master-worker computational model. In the above example, if an image is split into two sub-images, `getSplitSize()` would return 2. Finally, the `getObjectSize()` method simply returns the integer size of the piece of work in terms of its atomic units, i.e., an object of minimum granularity will have a size of 1.

For branch-and-bound applications, the methods in `BNBSplittable` do the following: `getMaxDepth()` returns the maximum depth of the search tree;
getAtomicDepth() returns the current atomic depth of a Splittable object (see Chapter 5 for a more detailed discussion of this parameter); and finally, incAtomicDepth() is used to increment the atomic depth of a task during the eager scheduler's dynamic depth expansion process.

### 7.2 A Generic Client & Host

We now demonstrate how a generic Javelin 3 application might look. Here, our example consists of two classes, a client and a host class. In this case, the client does not take part in the actual computation; it simply initializes the computation and handles results. Another way to program the application would be to provide a single class for both, as shown in the TSP example at the end of the chapter, where the client also performs as a compute engine.

The client must pass the fully qualified Java name of the host class into the JavelinClient constructor; create a root work object of the application class that implements the Splittable interface; obtain a reference to the local data manager by calling a static method, JRuntime.getDM(); pass the work to the data manager; set up result and done listeners; and finally, give the signal to begin the computation by calling JavelinClient.begin(). The
client skeleton is given below:

```java
import javelin.*;

public class GenericClient
    implements ResultListener, DoneListener
{
    private JavelinClient jClient = null;
    private JDataManager dm = null;
    private Splittable work = null;

    // Constructor, takes broker IP address as parameter.
    public GenericClient(String broker)
    {
        // Instantiate JavelinClient with
        // name of host class and broker address.
        jClient = new JavelinClient("javelin.demos.GenericHost",
                                   broker);

        // Create a work object of the class
        // that implements Splittable.
        work = new ...;

        // Obtain reference to the data manager.
        dm = JRuntime.getDM();

        // Pass the work to the data manager.
        dm.addWork(work);

        // Set client itself as result and done listener.
        dm.setResultListener(this);
        dm.setDoneListener(this);

        jClient.begin(); // Begin execution phase.
    }

    // ResultListener implementation.
    public void handleResult(Object result)
```
The host class must implement the Runnable interface, since the Javelin 3 host
daemon executes the host application as a thread. Therefore, the programmer
must implement the run() method, which is the first method that is invoked
on the host. The host constructor must be left empty, because the daemon
will call a default constructor for the host.

Prior to the computation, the host must obtain a reference to the local
data manager. Like the client, it does so by calling a static method,
JRuntime.getDM(). Then, the host starts the computational loop: ask data
manager for work, compute work, and return result. Once the data man-
ger returns null, indicating that there is no more work, the host terminates
by simply returning from the run() method. The host skeleton is presented
below.

```java
import javelin.*;

public class GenericHost implements Runnable {
    JDataManager dm = null;

    // Host constructor must be left empty,
    // since host daemon will execute generic constructor.
    public GenericHost() {
    }

    // All host initialization goes here.
    private void init() {
        // Obtain reference to the data manager.
        dm = JRuntime.getDM();
        ...
    }

    // Run method is called when host thread is executed.
    public void run() {
        // Explicitly call init() here.
        init();
        ...
        // Computational loop.
        while ((Splittable work = dm.getWork()) != null) {
            Serializable result = doWork(work);
        }
    }
```
dm{return}Result(result);
}
}

// Host computation method.
private Serializable doWork(Splittable work)
{
    ...
}
}

7.3 TSP Example

To further illustrate the use of our API, we now give code extracts from our TSP application. The two main classes are JTSP and TSPNode. The former is the executable class for both client and hosts, the latter is the implementation of the javelin.BNBSplittable interface. The JTSP class has to implement the Runnable interface, since the Javelin 3 host daemon is going to execute the host application as a thread. Therefore, the programmer must implement the run() method, which is the first method that is going to be invoked on each host.

We begin with extracts from the combined client and host class JTSP. To save space and increase readability much of the error handling code and some obvious type casts have been omitted.
import javelin.*;

public final class JTSP
    implements Runnable, ResultListener, DoneListener
{
    // local variables
    private static String broker = null;
    private static boolean isClient = false;

    // input parameters: graph & root node
    static Graph graph = null;
    static int root = 0;

    // local bound object
    private static TSPBound bound = null;

    // Javelin 3 variables
    static JDataManager jDM = null;
    private JavelinClient jClient = null;

    // ResultListener implementation,
    // called by bound propagation
    public void handleResult(Serializable result)
    {
        TSPBound newBound = null;

        synchronized(bound){
            if (result.getWeight() < bound.getWeight()){
                // set new bound and copy it,
                // so the lock can be released
                bound = (TSPBound)result;
                newBound = new TSPBound(bound);
            }
        }

        if (newBound != null){
            // invoke asynchronous bound propagation
jDM.propagateValue(newBound);
}

// DoneListener implementation
public void workDone()
{
    if (isClient){    // print out result
        synchronized(bound){
            System.out.println(bound.toString());
        }
    }

    jClient.terminate();
}
else {
    // host cleanup goes here!
}

// Constructor, should be left empty because
// default constructor is called by Class.newInstance()
public JTSP()
{
}

// necessary initialization goes here!
private void init()
{
    if (isClient) {
        // find out class name
        String className = this.getClass().getName();

        // create JavelinClient object
        // with class name, broker IP, and input graph
        jClient = new JavelinClient(className, broker, graph);

        // instantiate local bound object

bound = new TSPBound();

// obtain reference to data manager
jDM = JRuntime.getDM();

// set application type to branch & bound
jDM.setBNB();

// initialize with first chunk of work
TSPNode rootNode = new TSPNode(root);
jDM.addWork(rootNode);

// set result and done listeners
jDM.setResultListener(this);
jDM.setDoneListener(this);

// register with broker and start work.
jClient.begin();
}
}
else { // this is the host part!

// get input graph
// (Host provides a static method for this!)
graph = (Graph)Host.getInputData();

// initialize bound object
bound = new TSPBound();

// obtain reference to data manager, // and set application type, result and done listeners
jDM = JRuntime.getDM();

jDM.setBNB();
jDM.setResultListener(this);
jDM.setDoneListener(this);
}
// run method of host thread,
// must be called explicitly on client
public void run()
{
    // call init method on client & host
    init();

    TSPNode node = null;

    // main computational loop!
    while ( (node = (TSPNode)jDM.getWork()) != null ){
        node.doWork(node.getID());
        jDM.returnBNBResult(null, true);
    }
}

// main method on client,
// creates application instance and calls run() method
public static void main(String [] args)
{
    // parse arguments
    broker = args[0];

    graph = new Graph();
    graph.read(argv[1]);

    isClient = true;

    // create application instance
    JTSP tsp = new JTSP();

    // explicit call to run()
    tsp.run();
}
Next, we demonstrate how this application implements the `BNBSplittable` interface. The `TSPNode` class shown below does most of the actual computation, mainly in its `split()` method, which is called by the data manager to expand interior nodes of the search tree, while the `doWork()` method is called by the application to process the remaining atomic sub-trees. The algorithm used in `split()` and `doWork()` is basically identical. It performs a depth-first exploration of the search tree, using a Kruskal Minimum Spanning Tree calculation to obtain a tighter lower bound. The only difference is that `doWork()` recurses until it reaches the leaves of the local subtree, thus computing a complete sub-task, whereas `split()` only does a single split of one node, creating new instances of `TSPNode` as a result.

```java
import javelin.*;

final class TSPNode implements BNBSplittable {
  // atomic depth parameter
  private int depth = 4;

  // local status information
  int id, pathLength, pathWeight;
  boolean[] visited = null;
  int[] path = null;

  // current MST, does not get serialized
  transient MST currentMST = null;

  // Constructor for root node
```
public TSPNode(int id) {
    this.id = id;
    this.pathLength = 1;
    this.pathWeight = 0;

    visited = new boolean[JTSP.graph.n];
    visited[id] = true;

    path = new int[JTSP.graph.n];
    path[0] = id;

    currentMST = new MST(null, this, JTSP.graph, id);
}

// Constructor for child node
public TSPNode(int id, TSPNode parent, MST parentMST, int edgeWeight) {
    this.id = id;
    this.depth = parent.depth;
    this.pathLength = parent.pathLength + 1;
    this.pathWeight = parent.pathWeight + edgeWeight;

    visited = new boolean[JTSP.graph.n];
    System.arraycopy(parent.visited, 0,
                    visited, 0, JTSP.graph.n);
    visited[id] = true;

    path = new int[JTSP.graph.n];
    System.arraycopy(parent.path, 0,
                    path, 0, parent.pathLength);
    path[pathLength - 1] = id;

    currentMST = new MST(parentMST, this, JTSP.graph, id);
}
int getID() {
    return id;
}

// node can be split if we have not reached atomic depth
public boolean canSplit() {
    return pathLength < depth;
}

public int getMaxDepth() {
    return JTSP.graph.n;
}

public int getAtomicDepth() {
    return depth;
}

// used by eager scheduler to increment atomic depth
public void incAtomicDepth(int depthIncrement) {
    depth += depthIncrement;
}

// split Object, called by JavelinDataManager -
// this is where the actual computation takes place!
public Splittable [] split() {
    Splittable [] out = new Splittable[JTSP.graph.n];

    // if we’ve found a cycle, compare with current optimum
    if (pathLength == JTSP.graph.n){
        if (JTSP.graph.a[id][JTSP.root] < Graph.INFINITY){
            pathWeight += JTSP.graph.a[id][JTSP.root];
        }
TSPBound newBound = null;

synchronized(JTSP.bound){
    if (pathWeight < JTSP.bound.getWeight()){
        // found new bound
        JTSP.bound.update(pathWeight, path);
        newBound = new TSPBound(JTSP.bound);
    }
}

if (newBound != null){
    // bound propagation: send to client & children
    JTSP.jDM.returnBNBResult(newBound, false);
    JTSP.jDM.propagateValue(newBound);
}

    pathWeight -= JTSP.graph.a[id][JTSP.root];
}
}

else {
    // init MST if necessary
    if (currentMST == null)
        currentMST = new MST(null, this, JTSP.graph, id);

    // visit all unvisited adjacent nodes
    for (int i = 0; i < JTSP.graph.n; i++){
        if ((JTSP.graph.a[id][i] < Graph.INFINITY)
            && (visited[i] == false)){

            // cut out symmetries: make sure 1 is visited before 2
            if (i == 2 && visited[1] == false) continue;

            currentMST.reset();
            currentMST.calcLowerBound(i, JTSP.graph.a[id][i]);
            // check global bound
            if (currentMST.lowBound < JTSP.bound.getWeight()) {

    //
out[i] = new TSPNode(i, this, currentMST,
JTSP.graph.a[id][i]);
}
}
}

return out;
}

// sequential algorithm to solve
// remaining, atomic nodes
public void doWork(int nodeID)
{
    // implementation largely identical to split(),
    // just works on local copy of arrays.
    ...
}
Chapter 8

Conclusion

Javelin 3 is intended to harvest unused machine cycles of networked computers for ultra-large, coarse-grained adaptively parallel applications. It will run equally well on large cluster machines and networked workstations, as long as the ubiquitous Java platform is installed. It supports both the classical master-worker model of computation, as well as a branch-and-bound model of computation in which master-worker can be seen as a true subclass. Javelin 3’s easy to use API shields the application programmer from all communication, resource allocation, and task scheduling details, as long as the application can be cast as master-worker or branch-and-bound.

We use demand-driven work stealing, integrated with an advanced form of
eager scheduling, to balance the computational load and achieve a high degree of fault tolerance and scalability. The TSP branch-and-bound application acts as a stress test of our system, because the computational load of tasks at the same depth in the search tree can vary from as small as less than 1 millisecond to as large as more than half the time to complete the entire computation. To cope with this computational load variance, the eager scheduler dynamically increases the depth parameter of a task, and hence, decreases its size. Our experience is that this new form of eager scheduling substantially improves the system’s performance: we can now solve much larger problems than before, and also obtain better speedups for problems that previously did not scale well.

Performance for our test graphs showed near ideal speedups through 256 hosts. When 512 were used, speedup began to taper off, but not until 1,024 hosts were used did speedup reduce substantially. We have no doubt that, with a larger problem instance (e.g., one that would take 256 processors 20 or more hours to complete) we would see higher speedups even using 1,024 or more processors. We believe that the principle impediment to better speedups is the relatively short running time using 1,024 hosts, which means that any large tasks not yet broken up by the eager scheduler will still prolong the
critical path.

The quantification of performance degradation due to host failures that our model predicts has been borne out by the measurements taken of the raytracing application. This model thus enables users to predict the performance of their application as a function of host failure rates, as long as the given assumptions remain valid.
Chapter 9

Future Work

Over the years, the Javelin 3 system has reached a high degree of maturity, providing a stable platform for very large experiments. Although we are very pleased with the current results, many interesting problems remain:

- Currently, there is no way to preempt a host working on a large task, even though the task might have already been sub-divided and rescheduled to other hosts. Even though the loss of parallelism from such a “busy unresponsive” host is not dramatic in large configurations, it would be desirable to regain the host’s compute power.
• For fault tolerance, the system can now only deal with host or link failures. If a processor continues to work, but delivers faulty results, the problem of fault detection and correction becomes more complex. A quorum-consensus mechanism or Byzantine fault detector would be a big improvement here.

• We believe that Javelin 3 can handle much larger problem instances and host sets than tested so far. Unfortunately, we would require a significantly larger allocation of computing resources at NPACT’s IBM Blue Horizon, or other large parallel machines, for such larger speedup measurements.

• The Javelin system architecture was geared from the beginning towards geographically distributed networks. By using Java as a platform for heterogeneity, and considering long latencies in the way computation is handled, we built the system in a way that it should perform well when combining far away resources. We thought of running experiments that combine the clusters in Paderborn and San Diego, and maybe some resources at UCSB. However, the biggest problem in such a setup would be co-scheduling: it is not possible to predict when exactly the batch
scheduler on Blue Horizon will run a job, and it is certainly not possible to reserve a partition on the Paderborn cluster at exactly the moment when the job is run in San Diego. What is needed here is an integration of Javelin with the resource allocation services of the Grid, e.g. using the Globus toolkit.

- From an application point of view, it would be highly interesting to combine our approach with some sophisticated approximation algorithms for the TSP, e.g. memetic algorithms. Moscato et al. [31, 19] have considerable expertise in this field, and have suggested combining our efforts. This would dramatically tighten the upper bound for the TSP, and could possibly lead to a new record in solving the largest known instances.

- We would like to expand our programming model beyond the current application classes master-worker and branch-and-bound. Finding new application classes for Global Computing would keep interest in this new field of computation high and justify the ongoing research efforts.

- From the beginning, it was our vision that the broker network should use market-based resource allocation schemes such as auctions; hosts should
be rewarded; and clients should pay for the resources they use. The current work focuses on the computational aspects of large-scale applications; however, we feel that it is now time to expand the role of the broker.

The first two problems are entirely on the system side, and can be tackled in the near future. The final two issues are also under our own control, but are more long-term research goals. The three problems in between are ready to be tackled by us, but require some form of cooperation with other institutions, so the time frame is more unpredictable.
Bibliography


1238, Munich, Germany, August 2000.


1999.

based Parallel Virtual Machines. In *Proceedings of the ACM 1998 Work-
shop on Java for High-Performance Network Computing*, Palo Alto, CA,
February 1998.