Metacomputing on Commodity Computers

by

Arash Baratloo

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 1999

Approved: _____

Zvi M. Kedem

To my family and friends who made my graduate school years enjoyable.

Acknowledgements

Special thanks go to my friend and advisor, Professor Zvi Kedem. I still remember the first time he called me into his office and asked "what do you want, and what are you planning to do?" That was over five years ago. Since that day I have learned much from him, and his continual guidance and support has made this thesis possible. Last week, in helping me to make an employment decision, he called me into his office and asked: "what do you want and what are you planning to do?" I look forward to our continuing friendship.

I am also grateful to my friend Professor Partha Dasgupta whose insights were the foundation of my work. His thoughts and suggestions have greatly improved the content and clarity of this dissertation.

In addition to Zvi and Partha, I am deeply indebted to my other committee members, Emerald Chung, Krishna Palem and Vijay Karamcheti, for giving me many good suggestions on how to present, position, and think about my work.

I raise my glass in thanks to some special people who made my graduate school years enjoyable. Of these numerous friends, I can only mention a few, although I salute them all. Three cheers to Fangzhe Chang, Raoul Daruwala, Ian Jermyn, Ayal Itzkovitz, Holger Karl, Ilya Lipkind, Fabian Monrose, Jose Moreira, Mehmet

Karaul, Dmitri Krakovsky, Toto Paxia, Rob Rahbari, Naftali Schwartz, Nish Shah, David Stark, Peter Wyckoff, and last but not least, Yuanyuan Zhao. Special thanks to Amir Salehi, Cheon Kim, and Makis Anagnostou who encouraged me to go back to graduate school. I would also like to thank my roommates Rodney Christopher and Greg McCaslin. I first moved in with Greg until I could find another place to live. Six years later, I am still there, and I am thankful to him for providing me with a "home" during these years.

I cannot give enough thanks to Karen Cullen. I wish I only had to thank her for her patience during the final months, but I needed her patience and understanding throughout these six years. I am very glad she is with me.

To my kind friends who reviewed this dissertation, and to Karen Cullen who corrected their mistakes, my best wishes. Their careful and constructive comments improved the overall clarIty aNd readaBiliTy oF this documEnt.

My biggest gratitude, however, goes to my family. This work is dedicated to my parents, Nasrin Rahbari and Ahmad Baratloo, for their love; to my sister, Moji Baratloo, who always believed in me; to Cliff Balch my brother-in-law; and to my one and only niece, Halleh Balch.

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; and by the National Science Foundation under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The advantages of utilizing networks of commodity computers as a platform to execute compute-intensive parallel programs are well known: commodity computers are relatively cheap, widely available, and mostly underutilized. For example, here at the Department of Computer Science at New York University, students have access (and can login) to over 200 workstations. At the same time, I have seen individuals use a single workstation to run programs that take hours to complete. These programs could have been parallelized and made to execute on the network of available workstations which would have completed in just a few minutes. But programs were not parallelized—this dissertation addresses the reason behind this.

Utilizing networks of commodity computers to execute parallel programs is not an original idea—much research has been devoted to this topic, and many software tools have been built for developing such programs. Given that there are

relatively small number of widely available parallel programs that run on networks of workstations, a valid question to ask is: *if the hardware is widely available (which it is) and if there are tools for building parallel programs (which there are), then why aren't most programs able to run on networks of workstations?* I claim that major contributing factors are the complexity involved in software development and the extra effort needed to execute such programs. That is, *existing software tools make the development and execution of distributed parallel programs possible but not always feasible*; as a result, the added complexity outweighs the gains.

This dissertation presents a set of techniques for making parallel programs easy to design, build, and execute on networks of commodity computers. Furthermore, it presents a series of software systems to validate the feasibility of these techniques.

1.2 Metacomputing on Networks of Workstations

Commercial realities dictate that parallel computations typically will not be given a dedicated set of homogeneous computers. A number of studies have shown that in most industrial and educational organizations, a majority of computers are idle at any given time. This makes networks of commodity workstations a "free" computing platform if computations can be performed on machines that would otherwise have been idle. Such computing platforms have the following characteristics:

- *Non-uniform processing speeds:* Organizations generally purchase and upgrade workstations incrementally and as needed. So, naturally workstations bought at different times will have different processing speeds.

- *Unpredictable behavior:* Networks of workstations are considered a free computing platform only when they are time-shared. Because of external factors, even a network of identical machines is in effect, not homogeneous—different machines exhibit different characteristics, such as observed processing speed and available RAM. More importantly, these characteristics can change over time in unpredictable fashion.

- *Transient availability:* The set of idle machines changes over time. Hence, the availability of a workstation to participate in a parallel computation is *transient*: a workstation may become available for use by others at any time, and it may retreat in the middle of a computation.

The above characteristics result from external factors that exist in "real" networks of workstations. However, the available software tools for developing parallel programs do not always address important issues that arise in "real" networks of workstations. Specifically, load balancing, fault masking, and adaptive execution of programs on a set of dynamically changing workstations are neglected by most programming systems. The neglect of these issues has complicated the already difficult job of developing parallel programs.

Users can not constantly monitor a network to determine the availability of transient machines and to arbitrate among the demands of multiple adaptive computations. Adaptive computations are those that can adapt to external changes in resource availability and internal changes in resource requirements. Typically, a resource management system is used for this arbitration. However, no existing resource management system (on commodity systems) is capable of managing

several adaptive computations written using different programming tools.

1.2.1 Challenges

The shortcomings discussed above must be addressed to allow the effective utilization of networks of workstations and to facilitate the proliferation of parallel programs that execute on such environments. Specifically, a comprehensive solution must address programmability, adaptivity, load balancing, and the dynamic management of adaptive programs.

Programmability

It is clearly beneficial not to force programmers to learn a completely new programming language. The challenge is to provide programmers with a set of high-level programming constructs that incorporates natural syntax and semantics to express parallelism. More importantly, programmers should be allowed to develop programs that are independent of the execution environment. This will allow programmers to use their knowledge of the problem they are trying to solve, and not the execution platform, to guide the parallelism of the program.

It is generally agreed that shared-memory systems are more intuitive to program than message passing systems. Relaxed (i.e., non-sequential) shared-memory consistency models were introduced for performance reasons. However, programs developed for relaxed shared-memory consistency models are difficult to write, and as Lamport has pointed out [85], difficult to prove correct since formal methods for program correctness assume sequential consistency. The challenge is to provide a well defined programming model that maintains the properties needed to argue

correctness, while allowing programs developed for that model to perform well on distributed platforms.

Adaptivity

The parallelism inherent in a specific problem is independent of the number of machine a program for that problem will execute on. So why should the number of machines, a number that in most cases is unpredictable, influence the parallelism of a program? The challenge is to execute programs on any (reasonable) number of machines that are available at the time of execution. Because most networked machines have transient availability patterns, computations should also be able to integrate newly available machines and tolerate the removal of others. Network and machine failures are a reality, hence, computations should be able to tolerate failures as well. Programs that dynamically adapt to changes in resource and machine availability are referred to as *adaptive*. A common weakness in most parallel programming tools is the lack of support for adaptive programs. This weakness must be addressed.

Load Balancing

The performance of a parallel program is dependent on evenly distributing computations among participating machines. While a static partitioning of computations might be effective for dedicated clusters of workstations, programs running on shared networks of workstations require dynamic load-balancing to overcome the unpredictable machine behaviors. The challenge is to load-balance computations dynamically, and to adapt the computation to the speed of available machines.

As previously stated, in order to execute multiple adaptive computations on a shared set of machines, a resource manager is necessary. The resource manager must monitor resources and communicate the availability (and unavailability) of transient machines to executing programs. At the same time, programs must communicate their (internal) resource requirements to the resource manager. These communications are necessary to make the arbitration of resources possible. Many popular parallel programming tools do not provide a functional interface for such communication to occur. As a result, resource managers are not capable of managing multiple programs written with different programming tools. This severely limits the execution of parallel programs on a shared network of workstations. The challenge is to provide a set of mechanisms to overcome this limitation.

1.2.2 Contributions

The contributions of this dissertation include a set of mechanisms to address programmability, adaptivity, load balancing, and dynamic management of adaptive programs. Furthermore, this dissertation presents two software systems that demonstrate the feasibility of such mechanisms on time-shared networks of workstations. Calypso is a parallel programming system and a runtime system designed for adaptive parallel computing on networks of workstations; it is presented in Chapter 3. Chapter 4 presents a resource manager called ResourceBroker. ResourceBroker is unique in its ability to manage adaptive programs that were not developed to have their resources managed by external resource managers. The

work on Calypso and ResourceBroker has resulted in several original contributions which are summarized below:

- *Calypso decouples the programming model from the execution environment:* programs are written for a reliable virtual shared-memory computer with unbounded number of processors, i.e., a metacomputer, but execute on a network of dynamically changing workstations. This presents the programmer with the illusion of a reliable machine for program development and verification. Furthermore, the separation allows programs to be parallelized based on the inherent properties of the problem they solve, rather than the execution environment.

- *Programs without any modifications can execute on a single machine, a multiprocessor, or a network of unreliable workstations.* The Calypso runtime system is able to adapt executing programs to use available resources: computations can dynamically scale up or down as machines become available, or unavailable. The runtime system implements a technique called *two-phase idempotent execution strategy* that allows parts of a computation executing on remote machines to fail, and possibly recover, at any point without affecting the correctness of the computation. Unlike other fault-tolerant systems, there is no significant additional overhead associated with this feature.

- *Calypso automatically distributes the work-load depending on the dynamics of participating machines.* The load balancing mechanisms extend self-scheduling with two techniques called *eager scheduling* and *bunching*, respectively. The result is that fine-grain computations are efficiently executed in

coarse-grain fashion, and faster machines perform more of the computation than slower machines. Not only is there no cost associated with this feature, but it actually speeds up the computation, because fast machines are never blocked while waiting for slower machines to finish their work assignments—they bypass the slower machines. As a consequence, the use of slow machines will never be detrimental to the performance of a parallel program.

- *The combination of aggressive shared-memory caching techniques with adaptive scheduling policies is used to efficiently implement the shared-memory virtual machine on a network of workstations.* While providing adaptive execution, fault tolerance, and dynamic load-balancing, the experiments indicate that the overhead due to these mechanisms are surprisingly small for medium- to coarse-grained computations.

- A set of novel mechanisms to allow the management of *adaptive parallel programs that were not developed to have their resources managed by external systems.* ResourceBroker has been implemented to demonstrate these mechanisms. ResourceBroker is the first system that can support adaptive programs written in more than one programming system, and has been tested using a mix of programs written in PVM [59], MPI [67], Calypso, and PLinda [77].

1.3 Metacomputing on the World Wide Web

The Internet is effectively connecting millions of mostly idle machines. Its latest reincarnation as the World Wide Web has greatly increased the Internet's potential for utilization, including its potential to be used as a gigantic computing resource.

Web browsers' abilities to load and execute untrusted Java applets in a secure fashion provide the low-level means for metacomputing on the Web. Now the challenge is to provide a comprehensive end-to-end solution for metacomputing.

The use of local area networks as a metacomputing platform has been explored for many years. Numerous research projects have aimed at this goal, and based on their success, attempts have been made to extend existing systems to the Web. But utilizing the Web as a metacomputing resource introduces new difficulties and problems. First, the Web invalidates many of the assumptions used in designing parallel programming environments for networks of workstations. For example, the Web lacks a shared file system and machines are not homogeneous. Second, no individual user has access-rights, or could possibly hope to have access-rights to every machine on the Web. As a result, users who control individual machines, or software agents acting on their behalf, must donate the use of their machines to others. Users that donate the use of their machines are referred to as *volunteers*.

1.3.1 Challenges

A comprehensive end-to-end solution for metacomputing on the Web must address the concerns of programmers, users, and volunteers.

Programmers

The ease with which programs can be developed and maintained is a primary concern of programmers. Since the set of machines available on the Web is unpredictable, and the machines and networks may fail at any time, Web-based programs must be able to adapt to changing execution environments as well as

tolerate failures. The challenge is to relieve programmers from unnecessary complexities by providing a set of high-level constructs for parallel programming, as well as mechanisms for adaptive and fault-tolerant execution of programs.

Users

Users are mostly concerned with the ease, correctness, and efficiency of program executions. The machines connected to the Internet are not under a single administrative control and do not have a shared file system. As a result, executing programs on the Web requires more effort than on local area networks. The challenge is to make executing programs on the Web as simple as executing on local area networks.

Volunteers

Simplicity and security are important objectives for volunteers. Unless the process of donating and withdrawing a machine is simple, it is likely that many would-be volunteer machines will not participate. Furthermore, volunteers need assurance that the their computers will not be compromised by executing programs written by "strangers." Thus, the challenge is to provide an infrastructure that enables volunteers to find, join, and leave computations easily, and that executes programs in secure fashion.

1.3.2 Contributions

This dissertation presents two systems called *Charlotte* and *KnittingFactory* which, in unison, provide a comprehensive solution for metacomputing on the Web. *Char-*

lotte, which is presented in Chapter 5, facilitates the development of parallel programs and provides a runtime system for adaptive execution of programs on the Web. The work on *Charlotte* has resulted in several original contributions, which are summarized below:

- *Charlotte* is the *first parallel programming system to provide one-click computing on the Web.* That is, without any administrative effort, volunteers from anywhere on the Internet can participate in any ongoing computation by a simple click of the mouse.

- *Charlotte* is the *first system for parallel computing that uses a secure language and executes in a secure sandbox environment.* Because the program is implemented entirely in Java without any native (non-Java binary) code, volunteers have the same level of trust in running *Charlotte* programs as they do in running any other Java applet.

- Existing contributions are leveraged in providing a *metacomputer on the Web.* The programming environment is conceptually divided into a virtual machine model and a runtime system. The virtual machine model, as presented to the programmer, provides a reliable shared memory machine. The runtime system implements this model on a set of unreliable machines.

- Previous work originally developed for networks of workstations is extended to deal with the dynamics of the Web. Three integrated techniques—*eager scheduling*, *two-phase idempotent execution strategy*, and *bunching*—are used for load balancing, fault masking, and efficient execution of fine-grain tasks.

Charlotte is an example of a system that utilizes Java applets to execute programs on the Web. To fully utilize the potential of Java applets, the limitations imposed by the Java security model must be addressed. Chapter 6 presents *KnittingFactory*, an infrastructure for executing Web-based programs in the presence of the Java security model. *KnittingFactory* can facilitate the execution of general user-applications and can extend the capabilities of higher-level software such as *Charlotte*. The contributions of *KnittingFactory* include mechanisms for the following services:

- A distributed directory service to assist in finding Web-based applications on unknown hosts. This service is unique in migrating most of the computation away from directory servers to client browsers. In addition, the directory service supports a non-uniform name space designed specifically to keep parallel computations localized by assigning volunteers to nearby computations.

- A middleware service for direct applet-to-applet communication. This service is unique in making it possible for applets of the same distributed session, which are executing on different machines on the Internet, to directly communicate and exchange information.

1.4 Outline of the Dissertation

This dissertation is organized as follows. The concept of metacomputer and the reliable virtual machine as presented to programmers are presented in Chapter 2. That chapter also presents a set of runtime mechanisms that realizes the reliable virtual machine on a set of unreliable commodity computers. Calypso is presented

in Chapter 3. Calypso is the first system to demonstrate the effectiveness of the mechanisms presented in Chapter 2 in implementing a metacomputer on networks of workstations. Chapter 4 presents a set of mechanisms and a resource manager called ResourceBroker to facilitate the execution of adaptive programs, e.g., Calypso programs, on networks of workstations. The design and implementation of *Charlotte* is described in Chapter 5. *Charlotte* leverages the code-mobility and secure execution of Java applets to extend the concept of metacomputing to the World Wide Web. *KnittingFactory*, which assists the execution of programs on the Web is presented in Chapter 6. Research efforts related to parallel computing on networks of workstations and the World Wide Web, as well as resource management systems, are presented in in Chapter 7. Chapter 8 concludes this dissertation.

Acknowledgments

Calypso [10] is a result of a joint research effort with Partha Dasgupta and Zvi M. Kedem. A collaboration with Mehmet Karaul, Zvi M. Kedem, and Peter Wyckoff resulted in *Charlotte* [15, 16]. The work on *KnittingFactory* [13, 14] was a joint effort with Mehmet Karaul, Holger Karl, and Zvi M. Kedem. ResourceBroker [12] is a result of a collaboration with Ayal Itzkovitz, Zvi M. Kedem, and Yuanyuan Zhao.

Chapter 2

Metacomputer, an Overview

2.1 Virtual Machine Model

Distributed multiuser environments consisting of commodity machines typically exhibit unpredictable characteristics and are prone to failure. As is evident by the small number of widely available parallel programs, it is extraordinarily difficult to exploit distributed multiuser environments directly or even with existing software tools. Effective support for such environments suggests treating the entire collection of machines as a metacomputer that would present a single, seamless interface to programmers and users. This allows programs to be developed for execution on a reliable virtual machine whose binding to physical resources may vary dynamically during the lifetime of any particular execution—programs that dynamically adapt to available resources at runtime are referred to as *adaptive*. Furthermore, to minimize the complexity of distributed program development, a shared memory programming model accessible from a familiar programming language is needed.

This chapter presents such a metacomputer. Sections 2.1.1 and 2.1.2 present an overview of the programming model and shared memory semantics, respectively. Section 2.2 presents a set of key techniques that enable implementation of this metacomputer. These techniques are designed specifically for adaptive execution of programs on unpredictable distributed multiuser environments. Moreover, these techniques have been implemented by two software systems, Calypso and *Charlotte*, which are presented in Chapters 3 and 5, respectively.

2.1.1 Parallel Programming Model

The programming model involves augmenting a familiar sequential programming language with simple constructs to express parallelism. Parallelism is expressed by *parallel steps* within sequential programs. Parallel steps consist of one or more jobs that (logically) execute in parallel. Parallel steps are generally responsible for computationally intensive segments of the program. By contrast, sequential parts of programs are referred to as *sequential steps* and they generally perform initialization, input/output, user interactions, etc. A parallel step can occur anywhere in the program, including inside another parallel step, which is referred to as *nested parallelism*. Nested parallelism is not supported in current implementations of Calypso and *Charlotte*; however, other researchers have extended Calypso with nested parallelism [73, 119].

Figure 2.1 illustrates the execution of a program with two parallel steps and three sequential steps. It is important to note that parallel programs are written for a virtual shared-memory parallel machine irrespective of the number of computers that participate in a given execution.

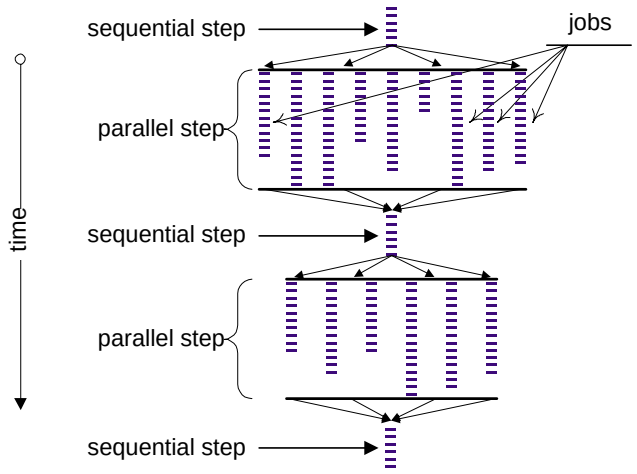


Figure 2.1: An execution of a program with two parallel steps and three sequential steps; the first parallel step consists of 9 jobs, the second parallel step consists of 6 jobs.

This programming model is sometimes referred to as a block-structured parbegin/parend or fork/join model [44, 107]. Unlike other programming models where programs are decomposed (into several files or functions) for parallel execution, this model together with shared memory semantics, allows loop-level parallelization. As a result, given a working sequential program it is fairly straightforward to parallelize individual independent loops in an incremental fashion.

2.1.2 Shared-Memory Semantics

Different shared-memory semantics can, and have been supported for the previously presented programming model. The current implementation of Calypso supports a generalization of the Concurrent Read Exclusive Write (CREW) programming model that allows a write operation to coexist with multiple read operations. In order to avoid race conditions, for instance when two jobs read and write the same

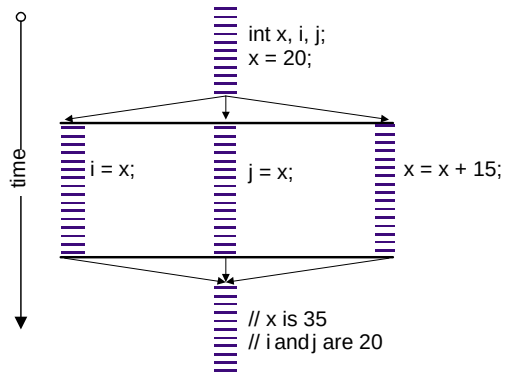


Figure 2.2: A parallel program with a single shared integer, x, and three parallel jobs. The first two jobs (from the left) read x and the third job writes x. Independent of the actual execution order, the first two jobs always read the value x contained at the beginning of the parallel step.

memory location, jobs execute in the (program) context in which they were created. As a consequence, read operations of unmodified data return the value of variables at the time the parallel step began. Furthermore, the effects of write operations are guaranteed to become visible at the completion of the parallel step, and not earlier.

Calypso's shared-memory semantics can be viewed as a generalization of the Bulk Synchronous Parallel (BSP) model [130, 84]. As is with Calypso, BSP guarantees that updates to non-local (i.e., shared) data locations become visible (to other processors) at the next superstep, although the updates might become visible earlier. In contrast to BSP, Calypso also guarantees that updates will not become visible earlier. As a result, a correct BSP program will execute correctly under Calypso's shared-memory semantics, but not necessarily the other way

round. The difference in memory semantics is illustrated in Figure 2.2, where three jobs concurrently read and write the same shared variable, x. The shared-memory semantics of Calypso guarantees that the read operations return the value of x at the beginning of the parallel step, i.e. 20. In contrast, using BSP's memory semantics the read operations could return either 20 or 35.

Charlotte supports *Concurrent Read Concurrent Write Common* (CRCW-Common) programs. This means that within a parallel step, one or more jobs can read a shared variable, and one or more jobs can write the same value to a shared variable. Similar to Calypso, results of write operations become visible at the completion of the parallel step.

2.1.3 Benefits

In addition of being isolated from the dynamics of the execution environment, the programs for the metacomputer have the the following properties:

In isolation execution semantics: The memory semantics ensure that jobs of a parallel step execute logically in isolation. Thus, (the source code of) a job can be developed and reasoned, independently of other jobs in that parallel step.

Proper Parallel Composition: A parallel program can be thought of as a composition of components. For instance, a parallel step is composed of jobs that semantically execute in parallel. Chandy *et al.* [36] defines *proper* composition as one in which the properties of the components are also the properties of the composed program. Proper composition is obviously helpful to reason about parallel

programs. The programming and shared-memory models for the metacomputer, as previously presented, preserve proper parallel composition.

Determinism: If each job of a (CREW) parallel step is deterministic, then the parallel step is deterministic as well; different executions of the parallel step produce identical results. This is clearly a desirable property, but many distributed shared memory systems do not provide such a guarantee—different executions of the same program could produce different results depending on the order of events. This is a specific weakness of systems supporting relaxed (i.e., non-sequential) shared-memory consistency models.

Independence of order semantics: As a consequence of *in isolation execution semantics* and *determinism* properties, the final result of a parallel step is independent of the order in which jobs were executed. In particular, it is possible for a sequential process to execute jobs comprising parallel steps in any order (while manipulating the memory to provide the in isolation execution semantics property) and the final outcome of the sequential execution will be identical to a parallel execution. Such a sequential process can be used to debug a deterministic program using standard debuggers for sequential programs, and programmers can be assured that once the program is correct, it will execute correctly on distributed environments.

2.2 Key Mechanisms

To execute parallel programs on networks of commodity computers, in many cases one assumes *a priori* knowledge of the number, relative speeds, and reliabilities of the machines involved in the computation. By having this information, the program can then distribute its load evenly for efficient execution. This knowledge can not be assumed for distributed multiuser environments, and hence, it is imperative that programs adapt to machine availability. That is, a program developed for a metacomputer must be able to integrate new machines into a running computation, mask and remove failed machines, and balance the work load in such a way that slow machines do not dictate the progress of the computation.

The traditional solution to overcome this type of dynamically changing environment has been to design and develop *self-scheduling* parallel programs. In self-scheduling programs, the computation is divided into a large number of small computational units, or tasks. Participating machines then pick up (in a self-service manner) and execute a task, one at a time, until every task has been executed. It is easy to see that faster machines generally do more of the work. For this reason, self scheduling has been used widely, and in the literature it is called the master/slave [59], the manager/worker [67] or the bag-of-tasks [33] programming model.

Self scheduling is a good starting point, but does not solve all the problems associated with executing programs on distributed multiuser environments. First, self scheduling does not address machine and network failures. Second, a very slow machine can slow down the progress of faster machines if it picks up a compute-

intensive task. Finally, self scheduling increases the number of tasks comprising a computation and, hence, increases the effects of the overhead associated with the process of assigning tasks to machines. Depending on the network, this overhead may be large and, in many cases, unpredictable.

2.2.1 Eager Scheduling

Calypso and *Charlotte* programs are implicitly self-scheduled. The basic notion of self scheduling is extended with two mechanisms initially proposed in [80]: *eager scheduling* (though this term was coined later) and *two-phase idempotent execution strategy* (TIES). Eager scheduling works in a manner similar to self-scheduling at the beginning of a parallel computation, but once the number of remaining tasks becomes less than the number of available machines, eager scheduling aggressively assigns and re-assigns tasks until all tasks have been executed to completion. Concurrent assignment of tasks to multiple machines guarantees that slow machines, even very slow machines, do not slow down the progress of a computation. Furthermore, if machines crash or become less accessible, for example due to network delays, the entire computation will finish as long as one machine is available for a sufficiently long period of time. Note that eager scheduling masks machine failures without the need to actually detect failures. In fact, failure is a special case of a slow machine (an infinitely slow machine). Eager scheduling is further discussed in Sections 3.5 and 5.4.

2.2.2 Idempotent Execution Strategy

Multiple executions of a program fragment (which is possible when using eager scheduling) can result in an incorrect program state. TIES ensures *idempotent memory semantics* in the presence of multiple executions. The computation of each parallel step is divided into two phases. In the first phase, modifications of the shared data region, that is the write-set of tasks, are computed but kept aside in a buffer. The second phase begins when all tasks have executed to completion. At that time, a single write-set for each completed task is applied to the shared data, and thus atomically updates the memory. Note that each phase is idempotent, since its inputs and outputs are disjoint. Put informally, in the first phase the input is shared data and the output is the buffer, and in the second phase the input is the buffer and the output is shared memory. The implementation of TIES in Calypso and *Charlotte* are described in Sections 3.5 and 5.4, respectively.

2.2.3 Bunching

The interplay of eager scheduling and TIES addresses fault masking and load balancing. *Dynamic granularity management* (or *bunching* for short) is used to mask network latencies associated with the process of assigning tasks to machines. Bunching extends self-scheduling by assigning a set of tasks (a bunch) at once. We have implemented Factoring [75], which computes the bunch size based on the number of remaining tasks and the number of currently available machines. Bunching has three benefits. First, it reduces the number of task assignments, and hence, the associated overhead. Second, it overlaps computation with communica-

tion by allowing machines to execute the next task (of a bunch) while the results of the previous task are on the network. Finally, bunching allows the programmer to write fine-grain parallel programs that are automatically and transparently executed in a coarse-grain manner.

2.2.4 Caching

When a machine picks up and executes a task, the shared-memory regions (i.e., virtual pages) that are accessed by the task are paged-in as needed. If any of the shared-memory pages is modified as the result of executing a task, the page is marked *dirty*, and the modifications to all dirty pages are flushed back to a memory manager process at the completion of the task. As a result, a software layer simulates a shared address space on machines that do not share physical memory. However, simplistic implementations of this software layer will lead to poor performance because of the high overhead associated with paging-in shared-memory pages over commodity networks. Caching techniques help to amortize this overhead by re-using paged-in regions whenever possible.

A parallel program for a metacomputer consists of alternating parallel and sequential steps. Hence, it is easy to associate virtual-step numbers with parallel and sequential steps, where the step numbers increase with the beginning and ending of each parallel step. When a machine pages-in a shared-memory region, it immediately creates a twin copy of the page and tags it with the virtual-step number. After completing a task, a machine compares shared-memory pages with twin copies to compute the differences that resulted from executing the task. These modifications are collated, tagged with the virtual-step numbers, and stored. The

structure of metacomputer programs allows an efficient caching technique as follows. Assume that the next task a machine is assigned to execute belongs to the parallel step of the previously executed task. In this case, the stored twin copies of shared-memory pages can be used to service page faults. Thus, within a parallel step, a machine will never need to retrieve a shared-memory page more than once. Now assume that the next task a machine is assigned to execute belongs to the next parallel step. In this case, the collated modifications are applied to twin pages to reflect the new state of shared memory. The updated memory can be used to service page faults if and only if no other machine modified the same page. Hence, shared-memory pages are kept valid and re-used as long as possible. An implementation of this caching technique is presented in Section 3.5.

2.2.5 Scheduling

A scheduling policy that assigns tasks accessing the same shared-memory pages to the same machine can reap the benefits of the previously described caching mechanism. Locality of reference can occur within a parallel step or across parallel steps.

Consider a data-parallel program, for example, a program that was parallelized by converting a for-loop into a parallel step. Within this parallel step, it is likely that the data set accessed by neighboring tasks (i.e., tasks created from consecutive iterations of the for-loop) accesses an overlapping set of shared-memory pages. This is particularly true for fine-grain tasks. For such cases, the scheduling policy is to assign neighboring tasks to the same machine—thereby reducing the number of page faults by taking advantage of the spatial locality of share data.

Now consider a larger program constructed by iterating over the parallel step described above. In such a case, the task that represents the iteration i of the for-loop will execute repeatedly and will access the same data set repeatedly. Thus, the second instance of task i has an affinity to the machine that executed this task before. For such cases, the scheduling policy reassigns continuations of the same task to the same machine in order to reduce the number of page faults across parallel steps—thereby taking advantage of the temporal locality of shared data.

Chapter 3

Calypso: Parallel Computing on Networks of Workstations

3.1 Introduction

Commodity networks of workstations (NOWs) can be considered an almost "free" computing platform for executing parallel programs, if time-shared with conventional interactive users. For example, large numbers of workstations exist in almost every organization, and as many studies [104, 127, 50, 101, 4, 39] have indicated, up to 60% of these machines are idle at any given time and can be used for other purposes. A platform composed of NOWs has been perceived by some as an inadequate substitute for a "real" parallel machine. Nevertheless, it is known that many applications run quite well on NOWs, and this has contributed to the popularity of systems such as PVM [59], MPI [67], Linda [33], and TreadMarks [2].

Challenges

Given that there are many compute-intensive problems that can execute on NOWs, a valid question to ask is: *why have programs that can make use of NOWs not proliferated?* A major reason is that the cost to harness NOWs is too high. That is, although NOWs are good value in terms of raw computing power (meaning hardware), the cost to harness this power (meaning software development and execution) still remains high.

Traditional distributed-program development systems focus on providing a toolkit, a set of function calls, or programming language constructs, but leave the programmer with the complex task of "pulling-it-all-together" in the form of a program. Such systems clearly aid program development, though inadequately. As evident by the small number of widely available programs that execute on NOWs, distributed-program development is too complex. This complexity results from the tight coupling of programs and their execution platforms: after all, programs execute on the available resources, however these available resources are not known at the time of development. Thus, it is beneficial if programmers can view NOWs as a single virtual metacomputing resource. Programs should be written for a clean and abstract model; and they should execute on networks of workstations utilizing resources as they become available.

Two general issues have to be addressed to effectively use NOWs as a parallel processing platform: making distributed application development easy enough for ordinary (non-specialist) programmers, and making executing such applications easy enough for ordinary (general) users. The challenging set of problems relat-

ed to providing a satisfactory solution for programmers is well understood. Such problems include programmability to simplify program development, high performance and scalability to make efficient use of resources, load balancing and fault masking to free the programmer from unnecessary programming complexity.

Contributions

Calypso is a software system specifically to assist programmers in developing parallel programs, and a runtime system to execute the programs on NOWs. Calypso is the focus of this chapter. Chapter 4 presents ResourceBroker, a resource management system to assist users in executing programs on NOWs. These two systems, in unison, provide an end-to-end solution for metacomputing on NOWs. The work on Calypso has resulted in several original contributions, which are summarized below:

- *Separation of Logical Parallelism from Physical Parallelism:* The parallelism expressed by a program should be independent of the parallelism provided by the execution environment, which is tied to the availability of workstations. Calypso separates the programming model from the execution environment. Programs are developed for a shared-memory virtual machine but execute on networks of dynamically changing workstations. The mapping between a program's parallelism and the execution environment is transparent.

- *Adaptivity and Fault Tolerance:* The Calypso runtime system is able to adapt executing programs to use the available resources, which may change over time. Calypso executions are resilient to failure. The Calypso runtime sys-

tem implements *two-phase idempotent execution strategy* that allows parts of a computation executing on remote machines to fail, and possibly recover, at any point without affecting the correctness of the computation. Unlike other fault-tolerant systems, there is no significant additional cost associated with this feature—in the absence of failures, the performance of Calypso is comparable to a non-fault-tolerant system. The impact of adaptivity on performance is discussed in Section 3.6.

- *Dynamic Load Balancing:* Calypso automatically distributes the work-load depending on the dynamics of participating machines. The load balancing mechanisms extend self-scheduling with *eager scheduling*. The result is that faster machines perform more of the computation than slower machines. Furthermore, because fast machines are never blocked while waiting for slower machines to finish their work assignments—fast machines can bypass the slower ones—this feature speeds up computations executing on machines of varying speeds.

- *High Performance:* The combination of aggressive shared-memory caching techniques with adaptive scheduling policies are used to efficiently implement the shared-memory virtual machine on networks of workstation. While providing the features listed above, the experiments indicate that the overhead is surprisingly small for medium- to coarse-grained computations.

- *Ease of Programming:* The programs are written in a language referred to as Calypso Source Language (CSL). CSL is essentially C++, with an added construct to express parallelism. The programming model is based on a shared

memory-model and is very easy to learn and use. Important aspects contributing to the ease of programming are the elimination of data partitioning and the need to specify how and when to move data between workstations. Section 3.2 presents the syntax and semantics of CSL.

Road Map

The rest of this chapter is organized as follows. Sections 3.2 and 3.3 describe how to write and compile Calypso programs, respectively. The steps involved in executing programs and the graphical user-interface are presented in Section 3.4. The implementation is presented in Section 3.5. Experimental results, in particular, the performance of Calypso programs for dynamic execution environments are presented in Section 3.6. Section 7.2 compares the Calypso system with other related work.

3.2 How to Write Calypso Programs

It is best to think of a Calypso program as a sequential program with embedded *parallel steps*. Sequential parts of a program, referred to as *sequential steps*, commonly perform initialization, I/O, user interactions, etc., whereas parallel steps are generally responsible for computationally intensive segments of the program. A parallel step is a new compound statement and it can be inserted anywhere in the program. It is important to note that Calypso programs are written for a virtual shared-memory parallel machine with an unbounded number of processors. This virtual parallel machine is realized by the Calypso runtime system. Therefore,

any program, irrespective of the number of parallel tasks, can run to completion on any number of machines.

A Calypso program basically consists of the standard C++ programming language, augmented by four additional keywords to express parallelism. These four key words are: `parbegin`, `parend`, `routine`, and `shared`. Shared memory semantics are provided for global variables that are tagged with the keyword `shared`. A parallel step starts with the keyword `parbegin` and ends with the keyword `parend`. Within a parallel step, multiple parallel *jobs* may be defined using the keyword `routine`. Completion of a parallel step consists of completion of all its jobs in an indeterminate order.

Example Program: Parallel Hello World

First, a Calypso program will be described through an illustration. Figure 3.1 contains a parallel implementation of a Hello World program. The program consists of three logical execution blocks: first is the sequential step up to the `parbegin`; second is the parallel step enclosed within `parbegin` ...`parend`; the rest of the program constitutes the third execution block, a sequential step. Let us consider the program in more detail.

- Notice the declaration of the global variable `array` in lines 5–7. The keyword `shared` is used to mark the shared memory regions. The runtime system guarantees data coherency (across multiple machines) only for shared regions of memory. Therefore, variables shared between two or more jobs, or between parallel and sequential steps, must reside in the shared region.

```
// file: helloWorld.csl
#include <ioastream.h>
#include <calypso.H>
const int size = 100;
shared {                              // declare the DSM region                  5
    int array[size];
}

void calypso_main(int, char *[]) {
    calypso_spawnWorker("sunra");  // start three worker processes              10
    calypso_spawnWorker("coltrane");
    calypso_spawnWorker("mingus");

    for (int i=0; i<size; i++)     // initialize the array
      shared->array[i] = -1;                                                   15

    int numberOfJobs;              // get the number of concurrent jobs
    cout << "How many jobs (at most 100)? ";
    cin >> numberOfJobs;
                                                                             20
    parbegin                       // in parallel, initialize the array elements
        routine[numberOfJobs](int totalJobs, int myId) {
           shared->array[myId] = myId;
        }
    parend;                                                                  25

    for (i=0; i<numberOfJobs; i++)
        cout << "Hello World from job number " << shared->array[i] << endl;
}
```

Figure 3.1: Parallel Hello World in Calypso.

- Analogous to the function named `main` in standard C++, Calypso programs begin by executing the `calypso_main` function, as in line 9.

- A Calypso program can spread its execution across multiple machines on a network. This is done by running *worker processes* on each remote host. Three methods are provided for spawning worker processes: Section 3.4 describes an interactive GUI, and Section 3.2 contains two library functions that can be used for this purpose. In lines 12–14 of this example, the library function `calypso_spawnWorker()` is used to spawn workers on three different machines, named `sunra`, `coltrane`, and `mingus`.

- In lines 14–15, the shared array is initialized with -1. Notice how the shared variable, in this case the `array` variable, is dereferenced with `shared->`. This is a general requirement. In lines 18–19, the user enters the number of parallel jobs, and this value is stored in the `numberOfJobs` variable. The program, thus far, is a standard C++ program.

- Lines 21–25 define a parallel step with only one routine. A parallel step begins with `parbegin` and ends with `parend` keywords. There can be one or more `routine` statements within a parallel step, and each routine statement defines zero or more parallel jobs. The statement `routine[numberOfJobs]` causes the expression inside hard-brackets to be evaluated at runtime and creates the specified number of jobs. This means that `numberOfJobs` syntactically identical jobs with identifications (Ids) `0` through `numberOfJobs - 1` will be created. The value of the expression along with the Id of each job are passed as formal parameters to each job. In our example, the formal parameters are

`totalJobs` and `myId`. In line 23, the jobs write their Id at an appropriate index of `array`.

- The final sequential steps in lines 27–28 produce the following output:

Hello World from job number `0`
Hello World from job number `1`
...

The Calypso Source Language

The programming language for Calypso is called the Calypso Source Language, or CSL. CSL is standard C++ with minor enhancements and several features that have certain structural constraints. This section describes each feature and each constraint in detail.

File Extension: CSL programs use `.csl` as their file extension. The Calypso preprocessor reads in a CSL program, expecting the `.csl` file extension, and writes a standard C++ program into a file with the same prefix name, but with a `.C` extension. For example, if `foo.csl` is input, the preprocessor will generate `foo.C`.

Calypso Header: Every CSL program must include `calypso.H`.

Main Function: As previously mentioned, the execution of a Calypso program begins with a function named `calypso_main`. This is analogous to C++'s `main` function.

Shared Variables: The Calypso runtime system provides the illusion of a shared-memory, parallel machine on a network of workstations. Shared-memory semantics is only provided for *shared variables*, i.e., variables that are tagged with the `shared` keyword. This implies that all (non-temporary) variables accessed inside parallel steps must be declared as shared. The shared memory coherence model is described on page 38.

The keyword `shared` which is used to declare a set of variables as shared, has the following syntax:

---

shared { optional member_list }

For example, the following code fragment declares integers `i` and `j`, and the character array `buffer` as shared memory.

```
shared {
  int i, j;
  char buffer[MAX_STRING_LENGTH];
}
```

Parallel Steps: A parallel step is a new compound statement that can be inserted anywhere in the program, except inside another parallel step. (Extensions to, and subsequent systems based on Calypso that supporting nested-parallel steps are briefly mentioned in Section 8.1). A parallel step starts at the keyword `parbegin` and ends with the keyword `parend`. There can be one or more `routine` statements defined within a parallel step. A parallel step generally has the following form:

```
parbegin
   routine [ integer−expr ] [ &foo1 , &foo2 ] (int num, int id) {
      // routine body 1
   }
   routine [ integer−expr ] (int num, int id) {                    5
      // routine body 2
   }
   routine {
      // routine body 3
   }                                                               10
parend (boolean−expr);
```

To better understand parallel steps, consider the events that occur as the result of the first `routine` of the above code segment. Logically, for each `routine` statement the following events occur.

- The (positive integer) expression `integer-expr` is evaluated and a total of `integer-expr` jobs are created and await execution. Each job will execute the same sequential code segment represented by *routine-body*.

- Jobs have access two arguments that have the same semantics as formal arguments of a function. In the above example, variables are called `num` and `id`, respectively. The variables are initialized to the number of processes created for the specific routine (i.e. `integer-exp`) and to the job Id. Job Ids are numbered: 0, 1, ..., `integer-exp` $-$ 1.

  Intuitively, the idea behind the `num` and `id` is for a job to control its behavior based on the number of jobs the `routine` expands to, and the current job Id.

- Before executing the job with Id j, $0 \leq j \leq$ `integer-exp` $-$ 1, the function

`foo1(int, int)` is called (with `integer-exp` and j as actual arguments) and is executed to completion. Similarly just after job j completes, the function `foo2(int, int)` is called with `integer-exp` and j as arguments, and is executed to completion. In essence, `foo1()` and `foo2()` behave as initializer and finalizer of jobs. The rationale behind supporting initializers and finalizers is to provide a general mechanism where by programmers can implement associated commutative operators. Examples of such operators are reduction, min/max, and sum operations.

The parallel step completes when either (1) all the jobs of a parallel step execute to completion, or (2) the `boolean-expression` evaluates to true, and then the execution continues with the first statement after the `parend`.

It should be noted that in the above example, the first `routine` statement illustrates the most general form; the second and third `routine` statements are special cases (syntactically simpler) and are handled by the preprocessor. At preprocessing time, the parallel step syntactically expands to the following:

```
parbegin
   routine [ integer−expr ] [ &foo1 , &foo2 ] (int num, int id) {
      // routine body 1
   }
   routine [ integer−expr ] [ void , void ] (int num, int id) {
      // routine body 2
   }
   routine [ 1 ] [ void , void ] (int, int) {
      // routine body 3
   }
parend (boolean−expr);
```

Shared Memory Programming Model: Within the body of a `routine` statement, the following applies:

- Both static and dynamic local variables can be accessed.
- Global shared variables can be accessed with the following restriction: within a given parallel step, any variable can be written by at most one job and concurrently be read by any number of jobs. That is, access to the global data is a generalization of Concurrent Read Exclusive Write (CREW) that allows a write operation to coexist with multiple read operations. Furthermore, each job executes in the (program) context in which the job was created. Thus, read operations of unmodified data return the value of variables at the time the parallel step began; results of write operations become visible at the completion of the parallel step.

- No other external effects are allowed. For instance, I/O statements are not allowed.

Library Functions

Two Calypso library functions can be called. Their purpose is to give the program the ability to request, by name or number, other host machines to join in the computation. The function prototypes are defined in `calypso.H` and are listed below.

---

calypso_spawnWorker(**char** *host);
calypso_spawnWorker(**int** num);

The `calypso_spawnWorker(char *host)` function call simply starts a shell on the (possibly remote) `host` machine, and spawns a worker process within the shell.

The `calypso_spawnWorkers(int num)` function requires the existence of a `.calypsorc` file in users' home directories. This function selects `num` names from the list machines' names in the `.calypsorc` file, and spawns worker processes on each of the machines.

3.3 How to Compile and Link Calypso Programs

A Calypso program may be written in one or more files with the `.csl` extension. A preprocessor translates those files into standard C++ programs stored in files with the `.C` extension. Each of the files can be compiled separately, and then linked with the Calypso library to produce an executable program.

For example, the following instructions preprocess, compile, and then link the

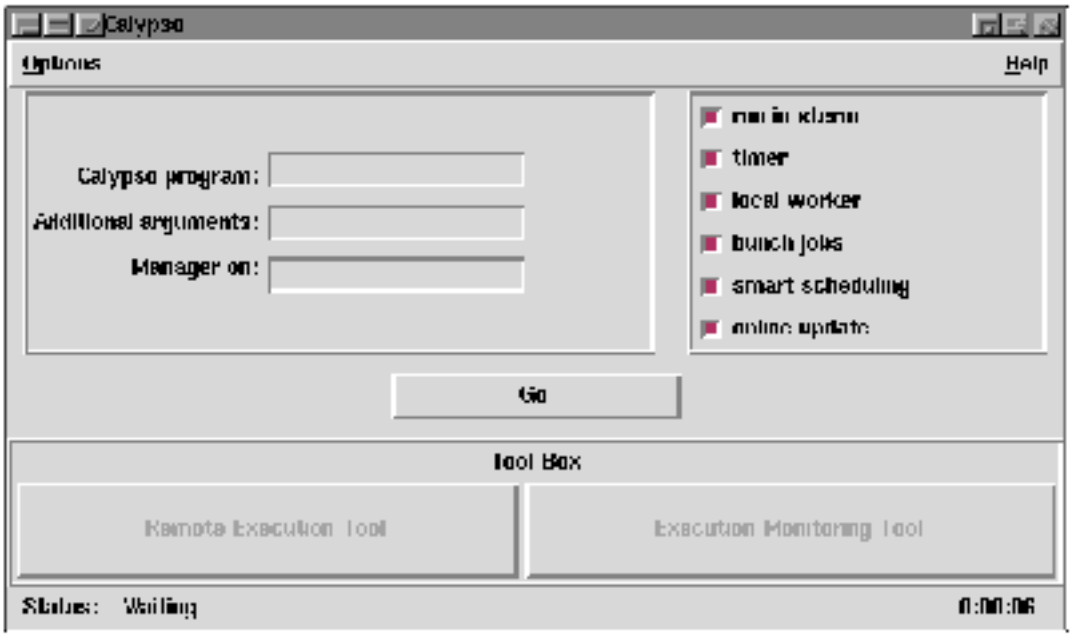


Figure 3.2: Screen snapshot of the Calypso Graphical User Interface

`helloWorld.csl` program seen earlier:

---

calypsopp helloWorld
g++ −c helloWorld −I$CALYPSO_ROOT/**include**
g++ −o helloWorld helloWorld.o −L$CALYPSO_ROOT/lib −lcalypso

The resulting program contains a runtime system that can adapt the program execution to use any number of machines made available to it. Furthermore, the runtime system balances the load among participating machines and can mask the failure of remote machines.

3.4 How to Run Calypso Programs

The simplest method of running a Calypso program is to submit the program to ResourceBroker for execution. ResourceBroker is a resource manager for dynamic allocation of resources to adaptive programs, and it is described in Chapter 4.

The Calypso Graphical User Interface (CGUI) provides an alternative method of running programs. It has an easy-to-use interface for specifying the calypso program, starting its execution, monitoring its progress, viewing the utilization of other computers on the network, and using them in the computation. The main window of CGUI is illustrated in Figure 3.4. In addition to entering the program name and optional arguments, users can use this window to select any of the following options:

- **Run in xterm:** runs the Calypso program within an xterm window
- **Timer:** measures program execution time
- **Local worker:** starts a worker process on the workstation running the Calypso program
- **Bunch jobs:** allocates a group of several consecutive jobs to each worker (see Section 2.2.3 on page 22)
- **Smart scheduling:** scheduler attempts to capitalize on spatial and temporal locality of shared data in assigning jobs to worker processes (see Section 2.2.5 on page 24)
- **Online update:** specifies whether the shared data is updated in real time (online) or once at the end of each parallel step

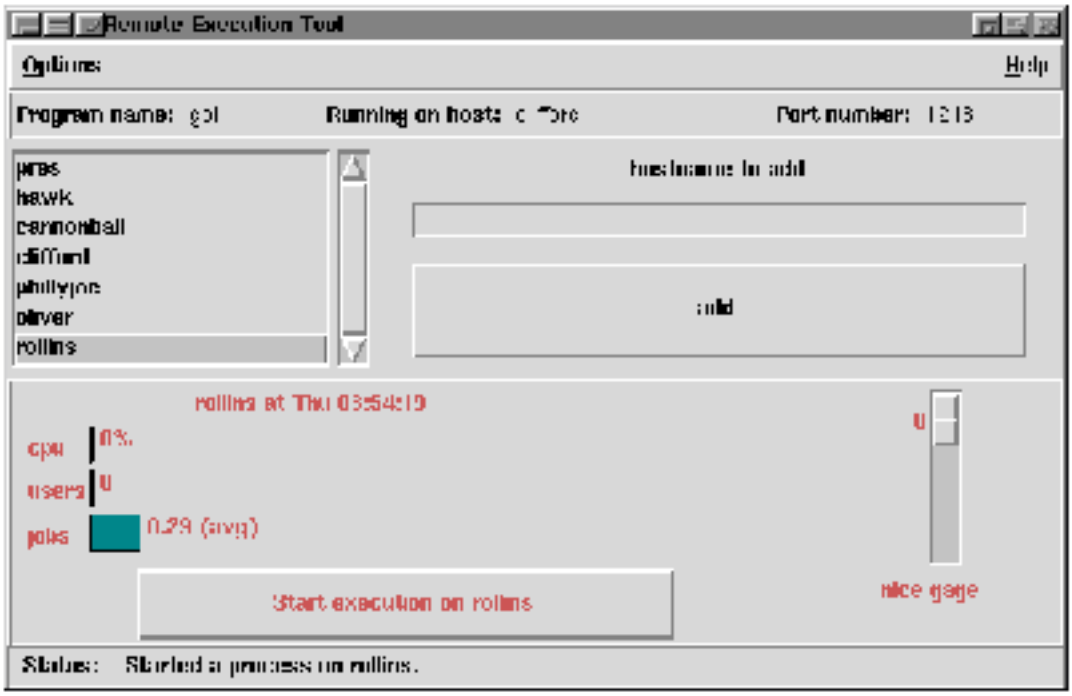


Figure 3.3: Screen snapshot of the Remote Execution Tool

The CGUI also features buttons to activate the *Remote Execution Tool* and the *Execution Monitoring Tool*.

Remote Execution Tool: The Remote Execution Tool is illustrated in Figure 3.4. With this tool, other computers on the network can be selected individually to participate in the computations. The utilization of other computers on the network can also be viewed. The *nice gauge* is used to set job priorities that can range from 0 to 20, where 0 is the highest priority. Other machine names, not included in the `.calypsorc` file, can be added in the *hostname to add* field.

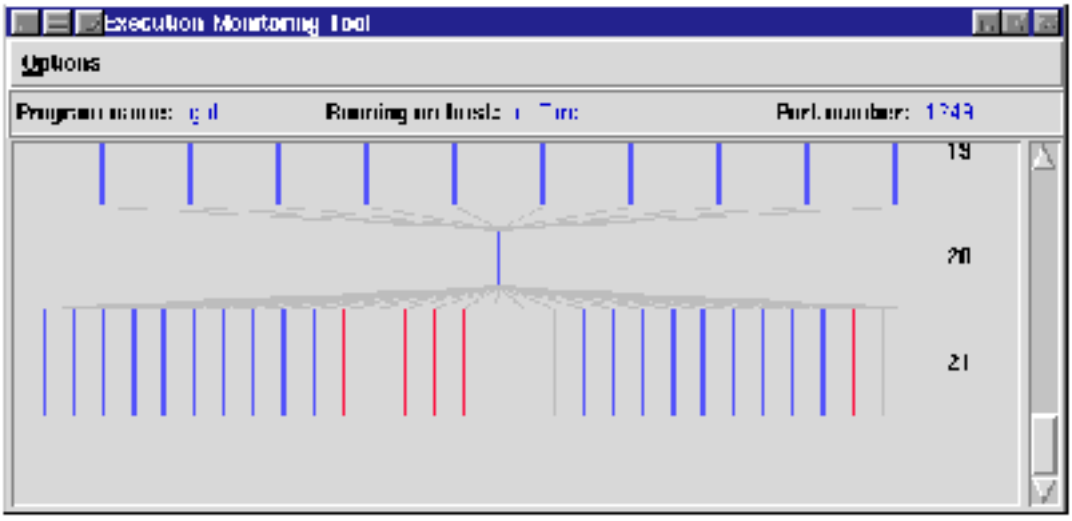


Figure 3.4: Screen snapshot of the Execution Monitoring Tool

Execution Monitoring Tool: Progress of a running program can be monitored by the Execution Monitoring Tool, which is illustrated in Figure 3.4. In this graphical tool, the assigned jobs are indicated as red lines. These lines change to blue when the job is completed.

3.5 Implementation

Preprocessing

A preprocessor takes a Calypso program and translates it into a standard C++ program. During the preprocessing stage the shared-data region is wrapped in a structure, and sequential code segments that define routines are stripped and wrapped into functions. Furthermore, the `parbegin`, `parend`, and `routine` statements are replaced with calls to functions implemented by the Calypso library.

```
shared {
  int  a[100];
  char b[50];
}
void calypso_main(int ac, char *av[]) {
  parbegin
    routine[100] (int num, int id) { shared->a[id] = id; }
    routine[50] (int n, int i) { shared->b[i] = '\0'; }
  parend;
}
```

(a) Pseudo program

```
typedef struct {
  int  a[100];
  char b[50];
} Shared;
Shared *shared = new Shared;
void calypso_main(int ac, char *av[]) {
  calypso_pt_initialize();            // replaces parbegin
  calypso_pt_addJob(100, &f1_0_0);  // replaces routine[100] (int num, int id)
  calypso_pt_addJob(50, &f1_1_0);   // replaces routine[50] (int n, int i)
  calypso_manageParallelJobs();      // two lines to replace parend
  calypso_pt_flush();
}
static void f1_0_0 (int num, int id) { shared->a[id] = id; }
static void f1_1_0 (int n, int i) { shared->b[i] = '\0'; }
```

(b) Preprocessed pseudo program

Figure 3.5: Relevant sections of a Calypso program before and after preprocessing.

These transformations are illustrated in Figure 3.5.

Execution Overview

A typical execution of a Calypso program consists of a central process, called the *manager*, and one or more worker processes, called *workers*. These processes can reside on a single machine or they can be distributed on a network. In particular, when a user starts a Calypso program, in reality, she is starting a manager. Managers immediately fork a child process that executes as a worker. It is important to note that managers and workers are executions of the same program image, and the memory layout of both processes are identical. A program runs as a worker when it is started with the `-calypso-worker <host> <port>` arguments; otherwise, it runs as a manager.

The manager is responsible for the management of the computation as well as the execution of sequential steps. The current Calypso implementation only allows one manager, and therefore it does not tolerate the failure of this process.

The computation of parallel jobs is left to the workers. There can be zero or more workers present at any one time, but at least one active worker is needed for a parallel computation to proceed; otherwise the computation is suspended, until a worker appears.

In general, the number of workers and the resources they can devote to parallel computations can dynamically change in a completely arbitrary manner, and the program adapts to the available machines. In fact, the arbitrary slowdown of workers due to other executing programs on the same machine, failures due to process and machine crashes, and network inaccessibility due to network partitions

are tolerated. Furthermore, workers can be added at any time to speed up an already executing system and to increase fault tolerance. Arbitrary slowdown of the manager is also tolerated; this would, of course, slow down the overall execution though.

Manager Process

The manager is responsible for the management of the computation. This management includes the following services: *scheduling service*, *memory service*, and *computing service*.

Scheduling Service

Upon reaching a parallel step, the manager calculates the number of jobs each `routine` defines and populates a *dispatch table*. A simplified version of the dispatch table for the program fragment in Figure 3.5 on page 44 is shown below:

Step	Address	Instances	Id	Done	Assignments
2	&f1_0_0	100	0	yes	-
2	&f1_0_0	100	1	no	2
...	...	...	...	...	...
2	&f1_1_0	50	0	no	4
2	&f1_1_0	50	1	no	1, 3
...	...	...	...	...	...

To explain the entries, consider the last (non-empty) row of the table. The fields indicate that: (1) the program is executing step number 2; (2) the code for

this job resides at (function) address &f1_1_1; (3) the routine created 50 instances of this job; (4) this is job number 1; (5) the job has not completed; and (6) workers 1 and 3 are currently executing this job.

The dispatch table is used to track the progress of the parallel step and to schedule jobs for workers. As previously stated, jobs are assigned to workers based on a self-scheduling policy. The manager waits until a worker requests a job, and then gives the worker an unfinished job (or a set of unfinished jobs as explained below) to execute.

Notice that the manager has the option of assigning a job repeatedly until it is executed to completion by at least one worker. This is referred to as *eager scheduling* as defined earlier in Section 2.2.1. Extending self-scheduling with eager scheduling provides the following benefits:

- As long as at least one worker does not fail continually, all jobs will be completed, if necessary, by this one worker.
- New workers are easily integrated into the computation, even in the middle of a parallel step.
- A slow worker asks for jobs less frequently, and thus does less work.
- jobs assigned to workers that later failed are automatically reassigned to other workers; thus crash and network failures are tolerated.
- Because workers on fast machines can re-execute jobs that were assigned to slow machines, they can bypass a slow worker to avoid delaying the progress of the program.

In addition to eager scheduling, Calypso's scheduling service implements several other scheduling techniques for improved performance. *Bunching* (as introduced in Section 2.2.3 on page 22) is used to mask network latencies associated with the process of assigning jobs to workers. Bunching extends self-scheduling by assigning a set of jobs (a bunch) to each worker at once. This is implemented by sending the worker a range of job Ids in each assignment. The overhead associated with this implementation is one extra integer value per job assignment message, which is negligible. The benefits of bunching are described in Section 2.2.3.

Furthermore, the scheduling service attempts to assign jobs to workers so as to minimize (shared) page-faults. For example, within a parallel step it seems likely that the data set accessed by neighboring jobs (i.e. with consecutive job Ids) are mapped to an overlapping set of virtual pages. This is particularly true for data-parallel programs. Calypso's scheduling service assigns neighboring jobs to the same worker—thereby reducing the number of page-faults by taking advantage of the spatial locality of share data. A similar technique is used to reduce the number of page-faults across parallel steps—thereby taking advantage of the temporal locality of shared data. These techniques are implemented by Calypso's scheduling service and are described in Section 2.2.5.

Memory Service

Calypso implements a software-based distributed shared memory on NOWs for a well defined CREW programming model. Notice that multiple and possibly partial executions of jobs caused by eager scheduling will lead to an inconsistent memory state without special care. Furthermore, efficient program execution on commodity

networks, with relatively high latency and low bandwidth such as TCP/IP on Ethernet, relies on efficient caching techniques. The implementation of the memory service addresses these issues as follows.

A Calypso program consists of alternating parallel and sequential steps. A virtual-step number is associated with each of alternating steps and the numbers are incremented with the beginning and ending of each parallel step. A manager constructs and maintains a vector of timestamps, `LMT`, to indicate the *last virtual-step number a shared data page was modified*. This vector is initialized with zeros before a manager executes the main body of a Calypso program which is virtual-step number one. The specific use of the `LMT` vector is described in the next section.

A manager write-protects (using a Unix system call `mprotect()`) the shared pages—the pages on which all and only shared variables are located—before executing sequential steps. During a sequential step if a manager attempts to write to a shared page, the generated (Unix `SIGSEGV`) signal is caught and handled by the manager itself. The signal handler first updates the `LMT` vector then unprotects the shared page that caused the signal. This mechanism serves to maintain correct entries in the `LMT` vector at all times. Subsequent write operations to the same page will proceed undisturbed and with no overhead.

Before each parallel step, a manager creates a twin copy of the shared pages and unprotects the shared region. The memory management service then waits until a worker either requests a page or reports the completion of a job. The manager uses the twin copy of the shared pages to service worker page requests. The message that workers send to the manager to report the completion of a job also contains the

modifications that resulted from executing the job. Specifically, workers logically bit-wise XORs the modified shared pages before and after executing the job, and send the results (diffs) to the manager. When a manager receives such a message, it first checks whether the job has been completed by another worker. If so, the diffs are discarded, otherwise, the diffs are applied (by an XOR operation) to manager's memory space. Notice that the twin copies of the shared pages, which are used to service worker page requests, are not modified. The finalizer associated with the completed job is then executed, so that the finalizer executes in the context of the new updates. The memory management of a parallel step halts once all the jobs have run to completion, at which point the `LMT` vector is updated to reflect the changes. The program execution then continues with the next sequential step.

It is important to reiterate the benefits of this implementation scheme. First, it provides idempotent (i.e. exactly-once) memory semantics even in the presence of multiple job executions resulting from eager scheduling. The collating technique (buffering diffs, accepting the first update and discarding others) in fact implements a *two-phase idempotent execution strategy* as defined in Section 2.2.2. As a consequence, program correctness is assured in spite of the multiplicity of executions. Second, by exploiting the structure of a CREW programming model, logical coherence and synchronization are both provided while false sharing is avoided.[1] Notice that different jobs can read and modify different regions of the same shared page of memory without causing page-shuttling. Third, jobs execute in the (pro-

[1] *False sharing* is when two concurrent processes need to access different partitions of a given page. Distributed shared-memory system needs to move this shared page back-and-forth between these two processes, causing what is called *page shuttling*.

gram) context in which the jobs were created since the memory modifications are not made visible until the completion of the parallel step. Hence, the final result is independent of the execution order. Finally, because jobs execute in isolation and maintain the *proper parallel composition* property, as defined by Chandy *et al.* [37], it is easy to prove the correctness of Calypso programs.

Computing Service

In addition to executing the sequential steps, the manager is responsible for executing the initializers and finalizers of jobs as previously described.

Worker Process

We now turn to the implementation of worker processes. Recall that workers are responsible only for executing jobs of parallel steps.

A worker establishes a TCP/IP connection to the manager at instantiation and maintains this connection throughout the computation. A worker repeatedly contacts the manager for jobs to execute. The manager sends the worker an assignment (a bunch of jobs) specified by the following parameters: the address of the function, the number of instances of the job, and a range of job Ids. The worker now executes this assignment for each of the job Ids assigned to it, as follows. The worker first access-protects the shared pages, and then calls the function that represents the current job. During this execution, the first time a worker accesses a protected shared variable a (Unix `SIGSEGV`) signal is raised (i.e. page-faults). The signal handler sends a request and fetches the appropriate page from the manager, installs it in the worker's process space, and unprotects the page for future use so

that subsequent accesses to the same page will proceed undisturbed. If the signal was raised because of a read operation, the computation proceeds; otherwise, for write operations the worker creates a twin copy of the page before proceeding. The execution of the function proceeds to completion. Then the worker identifies all the modified shared pages and sends the diffs—which are XOR of twin pages (which contains the before values) and the modified pages—to the manager. The worker then starts executing the next job in the assignment. Notice how bunching overlaps computation with communication by allowing a worker to execute the next job while the diffs are on the network heading to the manager.

Two optimizations have been implemented that improve the performance of Calypso computations, in particular, *caching* and *prefetching*.

Caching: Managers send the `LMT` vector to workers the first time a worker is assigned a job in a new parallel step. This vector consists of one integer for each shared page and it is piggybacked on a job assignment, hence, the associated network overhead is negligible. In addition to receiving manager's `LMT` vector, each worker constructs and maintains a similar vector that reflects, for each shared data page, the last virtual-step the worker had read and modified it. The `LMT` vector that workers receive from the manager contains the virtual-step number of the (program) context at which jobs should execute. The `LMT` vector that is constructed and maintained by each worker contains, for each shared page, the latest virtual-step number that the page is valid (i.e. not outdated) for that worker. Thus on page-faults, a worker can compare the two values for the page that caused the fault and *locally determine whether it needs to get a fresh and updated copy from the*

manager or use its local (cached) copy. So for instance, if a shared page was last modified in step number 4, it was read by a worker in step 6, and that worker is working on a job in step 8, then the worker does not fetch the page but accesses its cached copy.

This caching strategy is a low cost (almost free) strategy that produces significant performance benefits. Notice that shared pages that have paged-in by workers are kept valid as long as possible without a need for an invalidation protocol. Modified shared pages are re-fetched only when necessary. Furthermore, read-only shared pages are fetched by a worker at most once and write-only shared pages are never fetched. As a result, programmer does not declare the type of coherence or caching technique to use, rather, the system dynamically adapts. Invalidation requests are piggybacked on work assignment messages and bear very little additional cost.

Prefetching: Prefetching refers to obtaining a portion of the data before it is needed, in the hope that it will be required sometime in the future. Prefetching has been used in a variety of systems with positive results. A Calypso worker implements prefetching by monitoring its own data access patterns and page-faults, and it tries to predict future data access based on past history. The predictions are then used to pre-request shared pages from the manager. Depending on the regularity of a program's data access patterns, prefetching has shown positive results.

3.6 Experiments

A number of Calypso programs, including pattern matching, graphics, image processing, computational physics, scientific, and financial applications have been implemented. This section presents the performance the performance of several Calypso programs and compares the results with similar programs developed using other parallel programming systems.

We are interested in the behavior of a Calypso program on a network of workstations, which represents the "real world." Hence, the same program was executed several times in diverse and dynamic settings. In particular, experiments were conducted to analyze the following characteristics of Calypso programs:

1. The performance in an ideal execution environment when there are no failures, slow-downs, nor any need for load balancing or fault tolerance.

2. The contribution of load-balancing mechanisms to programs executing on various combinations of fast and slow machines.

3. The efficiency with which failures are masked, i.e., adapting to failures.

4. The efficiency of integrating additional machines into a running computation, i.e., adapting to a larger set of resources.

5. The ability of the runtime system to dynamically adapt a program execution to environments where some machines either die, become available, or suddenly slow at various times.

Machine Profiles

In order to experiment with heterogeneous machine speeds, ten machine *profiles* were defined. A profile determines a machine's behavior. Machine profiles are graphically illustrated in Figure 3.6 and further described below.

- *Machine A* is a perfect fast machine. It makes 100% of its CPU available to Calypso computations. This is a machine that does not fail or slow down during program executions. Machines with profile A model non-faulty workstations that are dedicated to parallel computations.

- *Machine B* is a non-faulty machine that is 50% slower than machines of profile A. The slowdown was achieved by running a high-priority background process; the slowdown was verified by ensuring that executions of a standard benchmark took 200% longer to execute with the background process running. Machines with profile B model slower, and maybe older, workstations.

- *Machines C*, *D*, and *E* are faulty machines that crash after 100, 200, and 300 seconds, respectively, from the start of a Calypso computation. This is done by killing the Calypso worker process manually at corresponding times.

- *Machines F*, *G*, and *H* are not available for Calypso programs at the beginning of a computation, but become available at a later time. Machines F, G, and H become available after 300, 200, and 100 seconds have elapsed form the start of the computation, respectively. This is achieved by starting Calypso worker processes manually at the corresponding times.

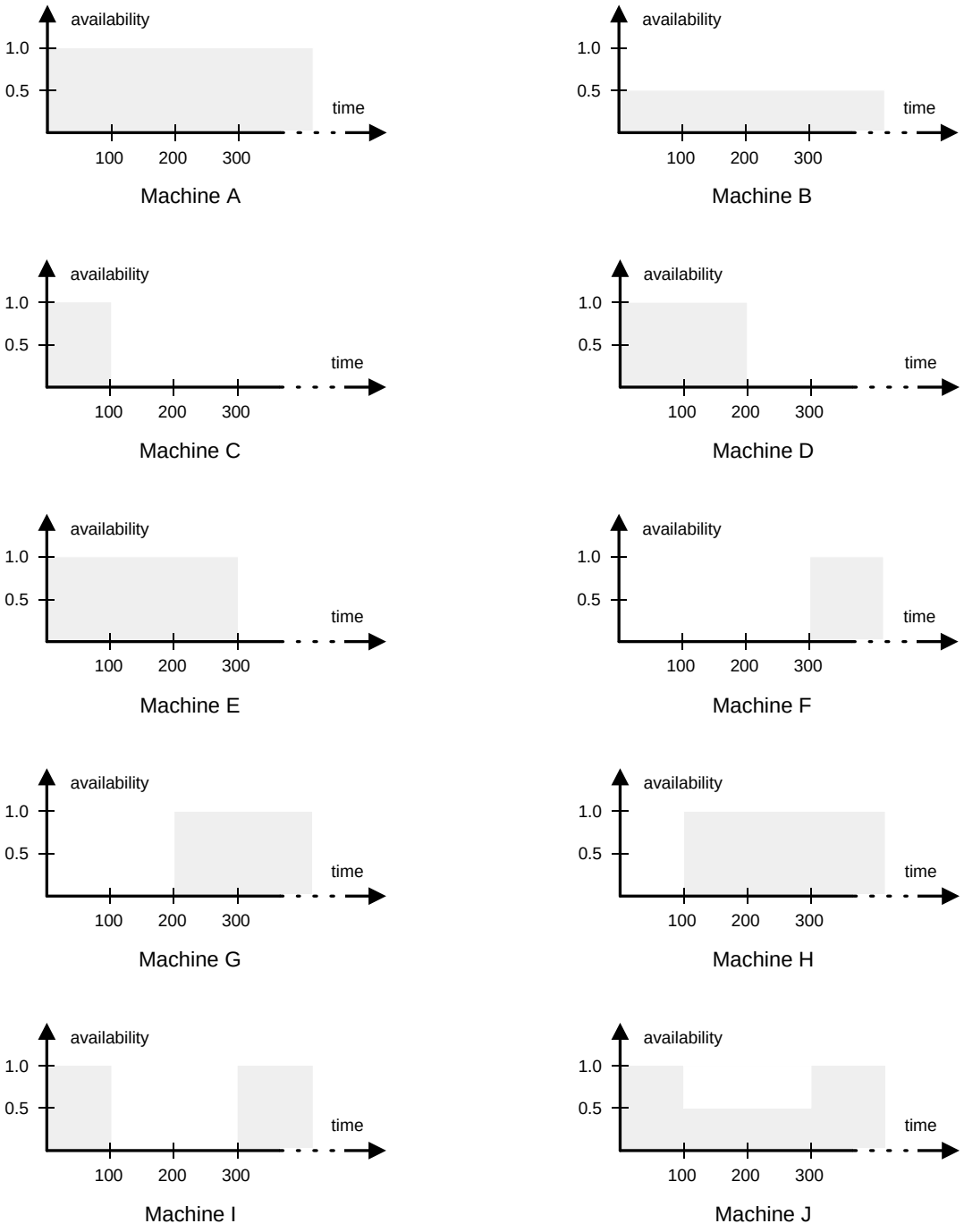


Figure 3.6: Profiles of worker machines.

- *Machine I* is a transient machine. For the first 100 seconds, 100% of its CPU cycles are made available for a computation, it then crashes and remains down for 200 seconds, at which time it recovers and continues to use 100% of its CPU cycles for the duration of the computation.

- *Machine J* models a shared workstation with fluctuating speed. For the first 100 seconds a computation, 100% of the CPU cycles of a machine with profile J is made available to the computation, then 50% for the next 200 seconds, and 100% for the duration of the computation.

At least one machine with profile A was used in each of the experiments. That machine ran a Calypso manager as well as a worker. Other participating machines ran workers.

Cost Model

This section describes the cost model used to report the performance experiments. Each machine with profile P is defined by the function availability$_P$. Availability is a function of time and depicts the fraction of the CPU resources made available to a computation. Thus, the availability of 1 denotes a machine that is fully available, and 0 denotes a machine that is unavailable. Then, if a computation lasts for time T, the *work* contributed by a machine is $\int_{t=0}^{T}$ availability$_P\, dt$. The work contributed by a machine is illustrated by the area of the shaded region for the time interval $[0, T]$ in each of the graphs in Figure 3.6.

In general, there will be several machines in a computation, say n machines with profiles, $P_1, \ldots, P_n$, respectively. If a computation lasts for time T, then the

total work is:

$$W = \sum_{i=1}^{n} \int_{t=0}^{T} \text{availability}_{P_i}\, dt.$$

The work W is the "charge" incurred by a computation for having machines available. Since machine availability is an external function, a computation is charged whenever a machine is available, whether the machine is effectively used or not. Given this charging method, it is obvious that the overhead includes the network time, the time wasted by redoing computations, the time taken to move data between workers and the manager, the time spent by the operating system, and other system activities.

Given work W, one can compute the number of equivalent perfect machines available to a computation. This is computed by:

$$\text{Number of Equivalent Perfect Machines} = \frac{W}{T}.$$

The interesting metric for users is the *speedup* of a parallel execution with respect to a sequential execution. The achieved speedup must be compared with the highest possible theoretical speedup given the same set of machines. In the absence of super-linear speedups, the number of equivalent perfect machines (as calculated above) is the upper bound on the obtainable speedup.

We now turn to computing the speedup of a parallel execution with respect to a sequential execution. The time it takes a sequential program to execute on a machine with profile A is referred to as $T_{\text{sequential}}$. If a parallel program executes in time T, then for that execution the speedup is given by:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T}.$$

Of course, as stated above:

$$\frac{T_{\text{sequential}}}{T} \leq \frac{W}{T}.$$

The closer the speedup is to the number of equivalent perfect machines, the better the performance of the system. To normalize speedup, *efficiency*, which is another measure of performance is used. Efficiency is defined as:

$$\text{Efficiency} = \frac{W_{\text{sequential}}}{W}.$$

Efficiency ranges between 0 and 100%: a value of 100% means that the execution achieved optimal speedup with respect to the best achievable. Efficiency measures how well resources that "happened to be available" are used in a computation.

Execution Environment

All experiments, unless noted, were conducted on up to 17 identical 200 MHz PentiumPro machines running Linux version 2.0.3, and connected by a 100Mbps Ethernet through a non-switched hub. The network was isolated to eliminate outside effects. In this environment, copying a memory page (4096 bytes) takes 5.4μs, bit-wise XORing two pages (creating diffs) takes 14.5μs, changing the protection of a page (using `mprotect()`) takes 4.4 μs, handling a (Unix `SIGSEGV`) signal takes 11.8 μs, round trip latency of a 32 byte message (a typical control message) takes 155.0 μs, and round trip latency of a 4096 byte message (a page) takes 881.4 μs.

Reported times are "wall clock" elapsed times, and not CPU or virtual times. It should be stressed that at the beginning of the measurements, workers did not have the shared data, and that at the end of the measurements, the manager had received and processed the modified data. Thus, overheads associated with

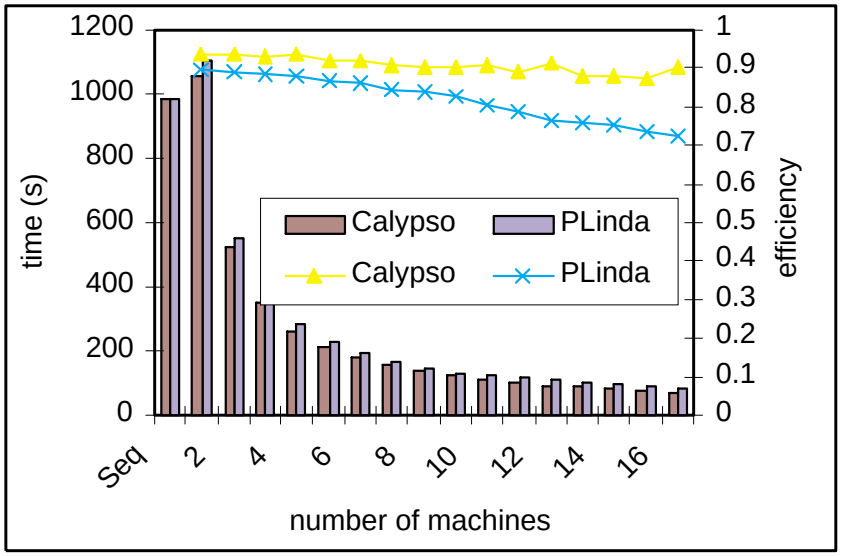


Figure 3.7: Scalability comparison of Calypso and PLinda biological pattern discovery programs.

networking, swapping, and updating shared data regions are included in the measurements.

3.6.1 Adapting to Failures

The first set of experiments presents the performance of Calypso and Persistent Linda (PLinda) [77] programs and compares how they can tolerate failures.

A biological pattern discovery program is used for this set of experiments. The program traverses a directed acyclic graph to find commonly occurring subsequences in sets of protein sequences. The parallelism is accomplished by concurrently traversing different paths of a graph. The PLinda program [89] implemented load balancing with an adaptive master process; the Calypso program was

a straightforward parallelization of the sequential program without explicit load balancing—the runtime system was responsible for load balancing. The results of executing these programs on 1 to 17 machines are presented in Figure 3.7.

The x-axis in Figure 3.7 shows the number of machines used and the label "Seq" indicates the execution of a sequential C++ program; the left y-axis reports the execution times; the right y-axis reports the execution efficiency as compared with the sequential run. The experiments show that a sequential program runs approximately 10% faster than a Calypso program on one machine. However, the Calypso program scales well with the number of machines as it is able to maintain its efficiency and outperforms the PLinda program by nearly 20% on 16 machines. Furthermore, the experiments illustrate that the load balancing provided by Calypso's runtime system is able to outperform the explicit load balancing implemented by the PLinda program.

PLinda adds fault tolerance to Linda programs by using light-weight transactions, whereas Calypso uses the combination of eager scheduling and two-phase idempotence execution strategy to mask failures. To measure the impact of different mechanisms used for fault tolerance, the same biological pattern discovery programs were executed on 16 machines, but in each run a process was forced to fail anywhere from 1 to 8 times. A failure consisted of crashing the process and immediately starting another process. The results are illustrated in Figure 3.8. The experiments show that the Calypso program is able to tolerate 8 process crashes and integrate 8 new processes into a running program with only 10% efficiency degradation. Note that PLinda's light-weight transactions tolerate crashes by detecting failed processes. However, efficient detection of failed processes is

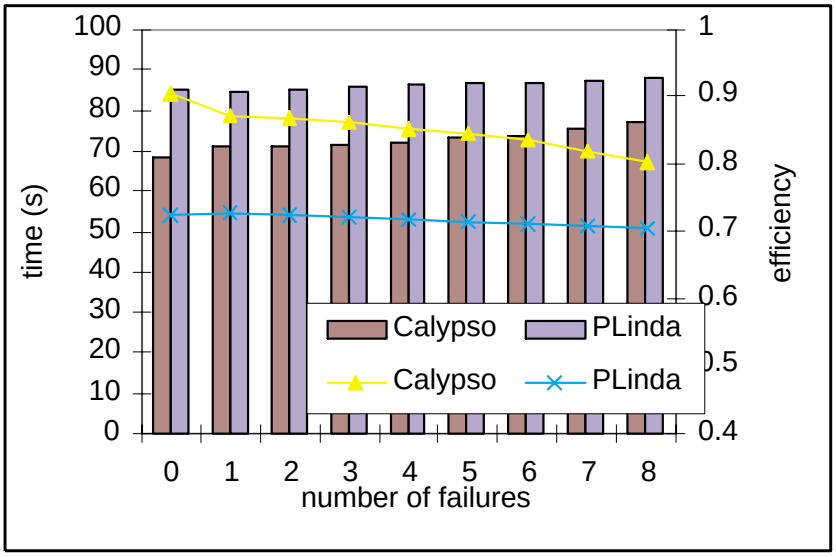


Figure 3.8: Comparison of Calypso and PLinda biological pattern discovery programs in the presence of failures.

not possible for all cases, e.g., in the case of network failures. Because of eager scheduling, Calypso is able to mask failures without detecting them, while being more efficient than light-weight transactions.

3.6.2 Adapting to Non-uniform Computing Speeds

The next set of experiments examines how a Calypso program performs when the computing speeds of participating machines are different, and compares the results with programs developed using CVM version 0.5 [128] and BSPlib version 1.4 [69, 70].

A standard matrix multiplication program that multiplies two 1024×1024 matrices filled with pseudo-random numbers using the trivial cubic time algorithm

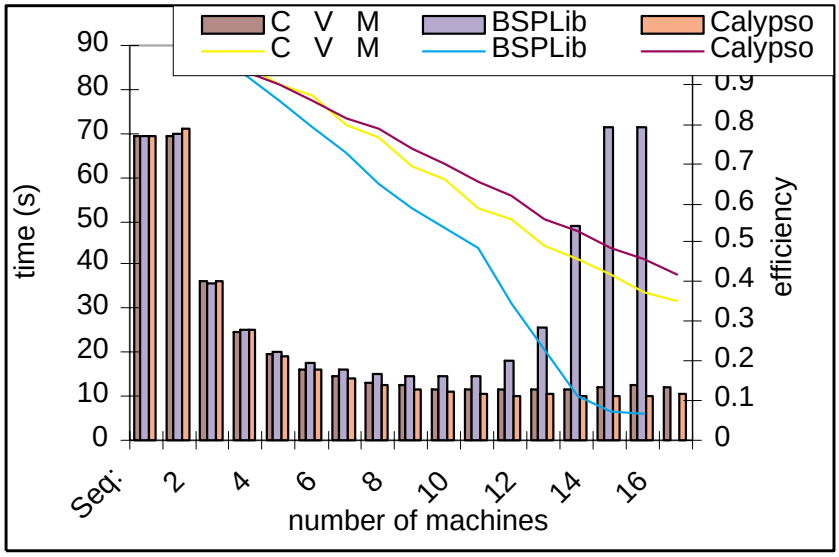


Figure 3.9: Scalability comparison of Calypso, CVM, and BSPlib matrix multiplication programs.

was used for these experiments. The Calypso program was parallelized using 1024 parallel tasks (independent of the number of machines it ran on). The CVM and BSPlib programs were parallelized using k processes, where k was set to the number of machines used in each execution; it should be noted that setting k equal to the number of machines produced the best performance for CVM and BSPlib programs. The results of executing these programs on 1 to 16 machines are presented in Figure 3.9.

Figure 3.9 shows that the Calypso program outperforms both CVM and BSPlib implementations. The BSPlib program does not scale well beyond 8 machines because of network contention. This network contention seems to have been caused by the implementation of the BSPlib communication library. Given that CVM is

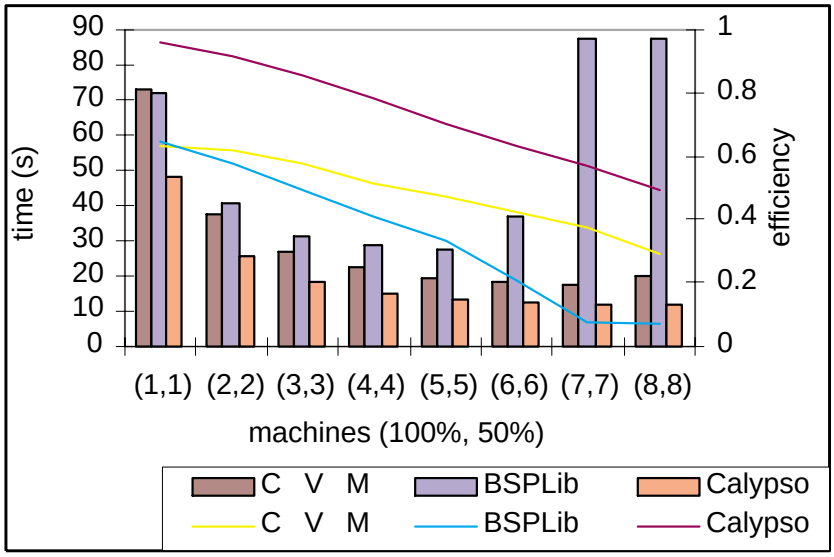


Figure 3.10: Comparison of Calypso, CVM, and BSPlib matrix multiplication programs running on a combination of fast and slow machines.

designed to demonstrate novel mechanisms for performance, whereas Calypso is designed to demonstrate novel mechanisms for adaptivity and fault tolerance, it is interesting to see that the Calypso program scales better than the CVM program. This is because bunching overlaps computation with communication, which results in computations continuously using the network (sending the results of fine-grained tasks) and, hence, better utilization of the network bandwidth.

The second set of experiments presents the performance of the same set of programs when the participating machines have different processing speeds. Specifically, the programs execute on 1 to 8 machines pairs, where each machine pair consists of one machine of profile A (perfect machine) and one machine of profile B (50% slower than the frist machine). The results are presented in Figure 3.10,

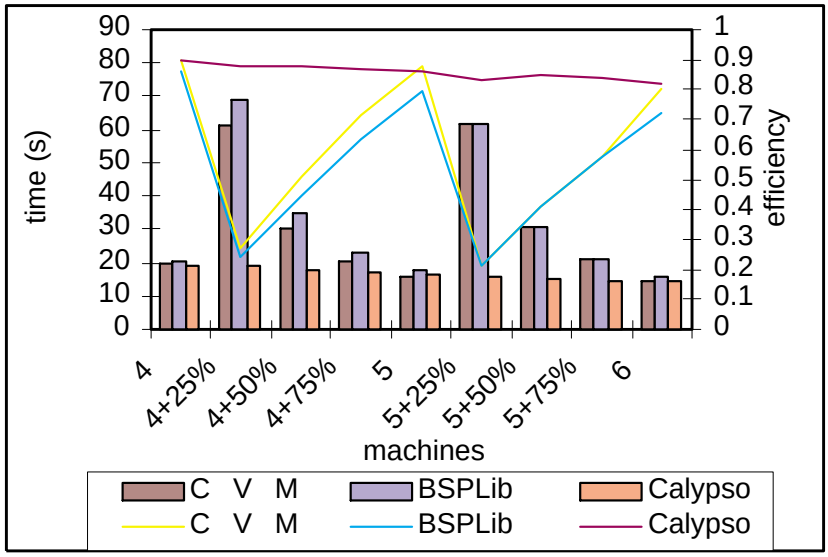


Figure 3.11: Comparison of Calypso, CVM, and BSPlib matrix multiplication programs running on non-uniform machine speeds.

and illustrate that for CVM and BSPlib programs, slower machines dictate the progress of computations. In contrast, the combination of self scheduling and eager scheduling allows slow machines to be used effectively in Calypso computations, while ensuring that they do not slow down computations.

The third set of experiments extends the previous experiments by further looking at the effects of non-uniform machine speeds. Specifically, the programs were executed on 4, 5, and 6 machines, but at most one of the machines was significantly slower than the rest. The results are shown in Figure 3.11. In the figure, the label 5 indicates the executions of 5 machines with uniform speed; and labels 5+25%, 5+50%, and 5+75% represent the executions on 6 machines where one of the machines was slowed down to 25%, 50%, and 75% the speed of the other 5

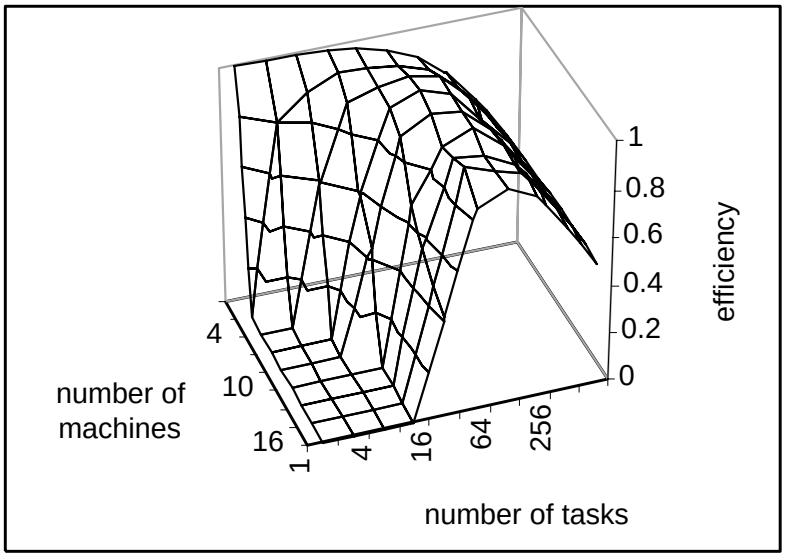
machines, respectively. The results indicate an addition of one slow machine can significantly slow down CVM and BSPlib computations, but this does not occur for Calypso computations. As expected, Calypso computations are efficient in using slow machines (compare the efficiencies of Figures 3.9 and 3.11), which results in predictable performance.

3.6.3 Adapting to Execution Environments

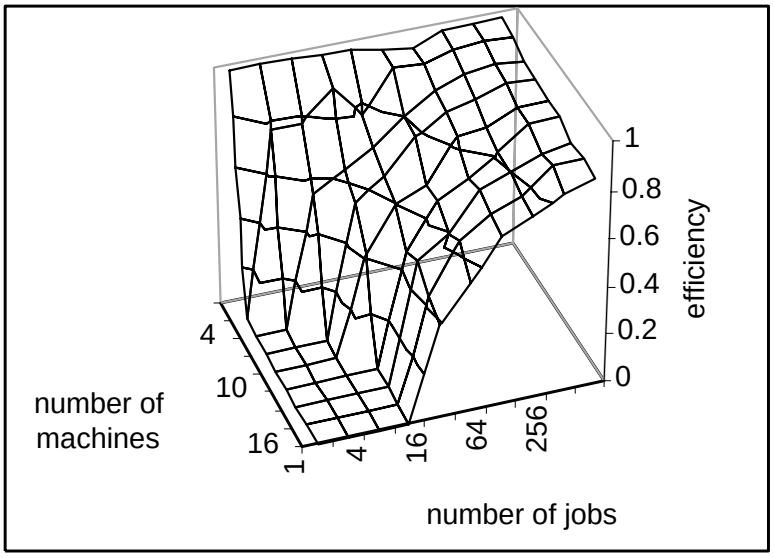
At the time of program development, a developer of a parallel program may not know the exact number of machines that will be used to execute the final program. Hence, it is important for programs to adapt to the number of machines they will execute on. This set of experiments examines the benefits of this type of adaptation. In particular, performance of Calypso and PVM [59] implementations of a ray tracing program using different degrees of parallelism and running on different numbers of machines is compared.

Ray trace is a graphical application for rendering and visualizing complex scenes. It implements a well known technique called *ray tracing* [49], which renders a scene (as seen) from a specific viewpoint. The algorithm "shoots" light rays from the position of the eye and traces their paths to compute the color and the brightness of the scene's pixels. There is obvious parallelism, and degrees of parallelism, across the rays shot through different pixels.

A publicly available sequential ray tracing program [41] was used as the starting point, then ported to Calypso and PVM. A 512×512 image containing only a single sphere was traced in every experiment. Figure 3.12 reports the efficiencies of Calypso and PVM implementations using 4 to 512 parallel tasks and running



(a) PVM



(b) Calypso

Figure 3.12: Comparison of Calypso and PVM ray tracing programs.

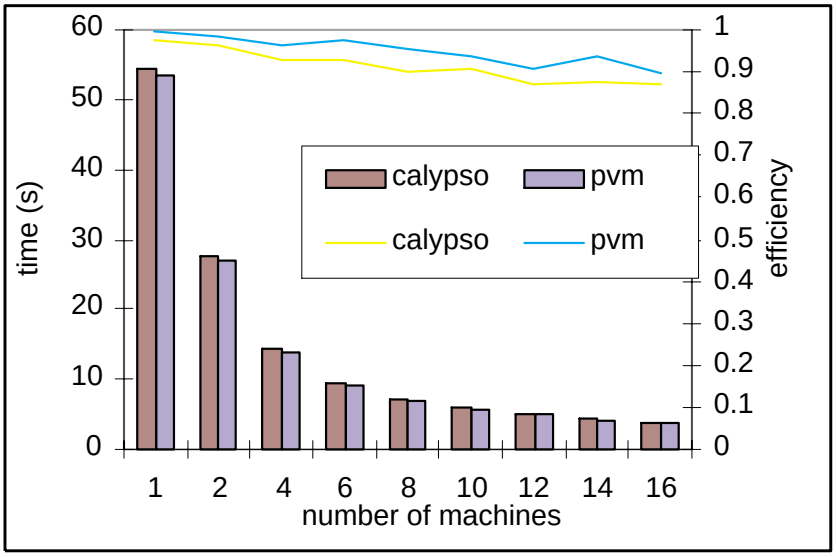


Figure 3.13: Scalability comparison of Calypso and PVM ray tracing programs.

on 1 to 16 machines. See Figure 3.12 (a). In the case of PVM, for a given number of machines the computation's efficiency peaks at a specific number of tasks. This is an expected result: for a fixed number of machines, too few tasks results in an imbalance of work loads, too many tasks introduces a contention spot at the manager. Furthermore, the overhead of task assignment overshadows the computation. The results, however, demonstrate that only the Calypso program with fine-grained tasks, e.g., 512 parallel tasks, adapts to different numbers of machines. This is illustrated in Figure 3.12 (b).

Figure 3.13 shows the performance of the Calypso program with 512 tasks and the performance of a PVM program with the optimal number of tasks. It is encouraging to see that the cost of dynamic adaptivity, load balancing, and fault tolerance mechanisms provided by the Calypso runtime system is at most 4%

compared to the optimal PVM program.

3.6.4 Adapting to Dynamic Execution Environments

A matrix multiplication program written in an older version of Calypso that did not implement bunching is used to demonstrate dynamic adaptivity of Calypso.

Setup

Experiments were conducted on up to 5 identical Sun SPARCStation SLC workstations in a public laboratory connected by a 10 Mbps Ethernet through a non-switched hub. The tests were run in the middle of the night in order to minimize external effects. Since we had no control over external effects, test results may underestimate the potential performance. The five workstations were disk-less and, hence, swapped on the network.

Performance Results

Five families of experiments were conducted. The results are graphed showing the total time, the achieved speedup, and the number of equivalent perfect machines (which sets the upper bound for the speedup). The performance graphs are labeled with machine profiles. For example, 3A+2D label indicates that a particular execution was charged for the work of 3 machines with profile A and 2 machines with profile D.

Ideal execution environment: The first family of experiments examines the performance of a Calypso computation using from 1 to 5 machines of profile A, which

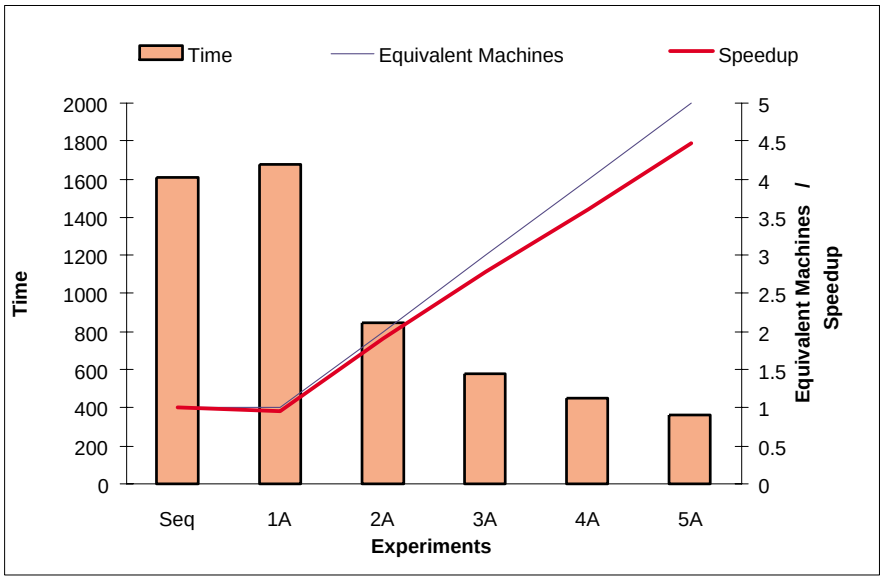


Figure 3.14: Performance of a Calypso program in an ideal execution environment.

devote all their resources to the computation.

The results are illustrated in Figure 3.14. The label "Seq" indicates the execution of a sequential C++ program. The comparison of the sequential execution time with the case 1A, where the manager and a worker were on one machine, indicates that 4% performance degradation is due to the Calypso runtime system. Figure 3.14 shows that the efficiency of the Calypso program ranges from 96% to 89%. It is encouraging to see that when there are no failures or slow-downs, Calypso bears little overhead, even though it is "prepared" to handle such adverse cases.

Adapting to imbalance of computing speeds: The second family of experiments examines how a Calypso computation performs when some of the machines are fast (profile A) and some are slow (profile B).

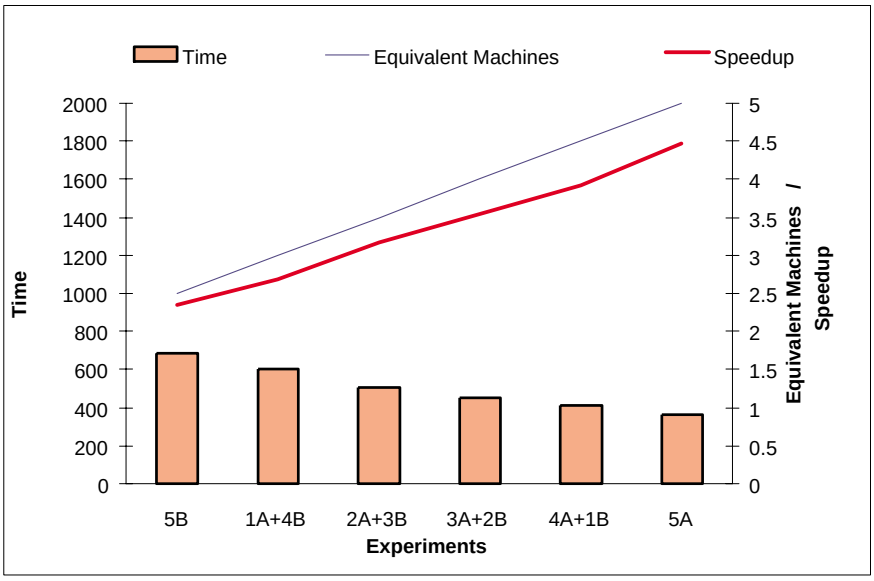


Figure 3.15: Performance of a Calypso program on a mix of fast and slow machines.

The results are illustrated in Figure 3.15. The label 5B indicates that 5 slow machines were used, the label 2A+3B indicates 2 fast and 3 slow machines were used. The speedups ranged from 2.34 for 5B (where the best possible speedup is $5 \cdot 0.5 = 2.5$), to 3.92 for 4A+1B. One case is worth discussing. Note that the speedup in case 4A+1B (3.92) is better than in case 4A (3.59) and poorer than in case 5A (4.47). This is in contrast to systems that do not provide automatic load balancing, the addition of one slow machine helped the 4 fast machines, rather than slowing down the computation.

Adapting to failures: The third family of experiments examines the effects of process crashes on a running Calypso computation. Three experiments were conducted. In each case, a Calypso computation was started on 5 machines, and 2 were crashed after 100 (3A+2C), 200 (3A+2D), and 300 (3A+2E) seconds, respectively.

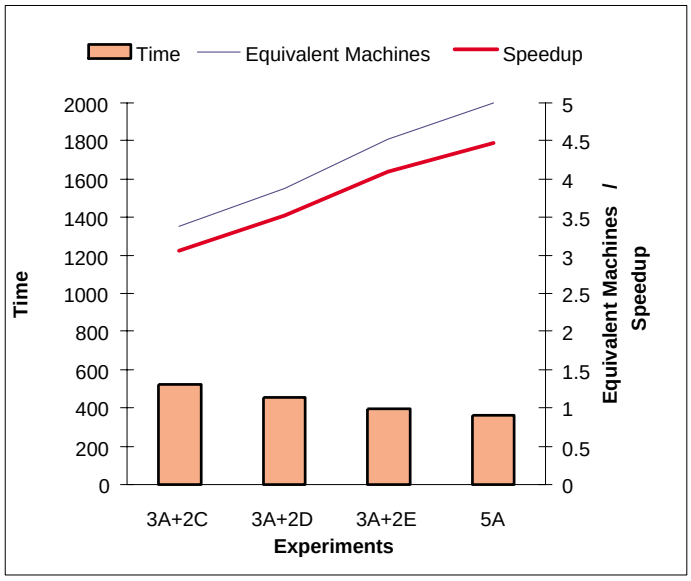


Figure 3.16: Performance of a Calypso program on a set of faulty machines.

The results are illustrated in Figure 3.16, which also reports 5A for comparison. The results indicate that speedups of 3.06 for 3A+2C, 3.51 for 3A+2D, and 4.11 for 3A+2E were achieved. Again, note the effective use of the C, D, and E machines: as the speedup for 3A+2E is better than 3A+2D, which is better than 3A+2E, and all three are better than just 3A (2.76). Thus the speedup increased monotonically with additional machine availability. The overhead remains quite low, as shown by the efficiency, which ranges from a high of 91% to a low of 90%.

Adapting to the addition of new resources: This family of experiments examines how well the Calypso runtime system can utilize workstations that become available after computations have begun. Three experiments were conducted. In each case a Calypso computation was initially started on three machines (3A). Then two additional machines joined the computation 100 (2F), 200 (2G) and 300

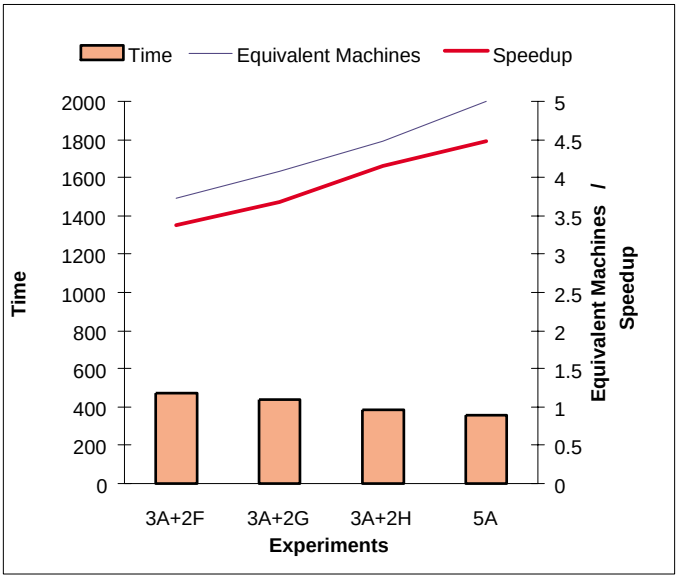


Figure 3.17: Performance of a Calypso program on a growing set of machines.

(3H) seconds later.

The results are illustrated in Figure 3.17. Again, the figure includes the case of 5 machines machines with profile A for comparison. The speedups are: 3.38 for 3A+2F, 3.68 for 3A+2G, and 4.15 for 3A+2H. Here, the efficiency ranged from a high of 92% to a low of 90%. These experiments show that the Calypso runtime system is able to effectively utilize machines that dynamically (and unpredictably) join a computation while in progress.

Adapting to dynamic execution environments: The final family of experiments illustrates the ability of the Calypso runtime system to adapt a program execution to an environment where machines crash, recover, slowdown, and speedup while the program is executing.

The results are illustrated in Figure 3.18. These experiments examine how a

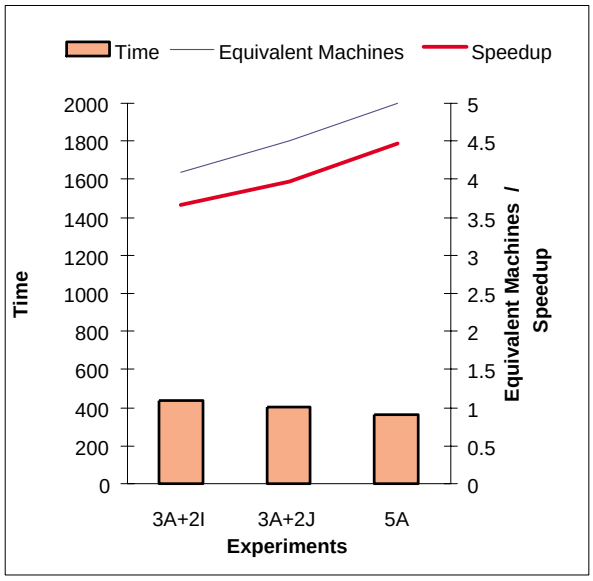


Figure 3.18: Performance of a Calypso program in a dynamic execution environment.

computation performs when 2 out of 5 machines (a) slow down to 0% and then recovered (3A+2J), and (b) slow down to 50% and then recovered (3A+2I). As the results indicate, the efficiency ranges from a high of 89% to a low of 88%. These experiments demonstrate the ability of our system to utilize resources as they become available, even if availability changes over time. The Calypso runtime system does not "give up" on machines that can still help the computation, even if they are slow, because such machines will not slow down the computation.

Chapter 4

Mechanisms for Just-in-Time Resource Allocation

4.1 Introduction

At the New York University's Distributed Systems Laboratory we have been developing and experimenting with *adaptive parallel programs* that run on networks of machines. Most master/slave PVM [59] programs, self-scheduling MPI [67] programs, bag-of-tasks Linda [33] programs, and all Calypso [10] programs are adaptive. For PVM, MPI, and Linda, programs must be written so that they are able to tolerate machine removals; whereas for Calypso, this service is provided by the runtime layer.

While working with adaptive programs we encountered a resource contention caused by having a number of users running a mix of sequential and parallel jobs. Interactive users were experiencing slow turnaround and non-responsiveness. Users

running parallel jobs were measuring widely-varied execution timings for repeated runs of the same program. This was somewhat unexpected, particularly since we have nearly twice as many machines as users: nearly half of the machines on our network have no monitors and are located in an isolated machine-room. Hence, we were anticipating that excess resources would eliminate contention. Our findings indicate that in order to achieve the 2:1 rule reported by Arpaci *et al.* [4], that "a NOW [Networks Of Workstations] cluster of approximately 60 machines can easily sustain a 32-node parallel workload in addition to the sequential load placed upon it by interactive users," intelligent allocation of resources is necessary.

In order to allocate, deallocate, and reallocate idle machines to running jobs, a *resource management system—resource manager* for short—is needed. Ideally, this service should be available to any compiled executable and should not require explicit programming. For efficient utilization of resources, machines should be allocated to computations only when requested and not pre-allocated and reserved. We refer to this ability as *just-in-time allocation of resources.* Furthermore, a resource manager should be able to simultaneously manage programs developed using different programming systems. As discussed in Sections 4.2 and 7.4, existing user-level resource managers for adaptive jobs do not support multiple standard parallel programming systems. This severely limits their applicability.

Contributions

The technical contribution of this chapter is a set of novel mechanisms for just-in-time allocation of machines to jobs. The mechanisms enable transparent management of *adaptive parallel programs that were not developed to have their resources*

managed by external systems. To validate the feasibility of these mechanisms, a resource management system called ResourceBroker has been developed. ResourceBroker is unique in providing a set of features previously not available in any system. These features include:

- *Just-in-time allocation of resources:* Machines are allocated to running jobs as their resource requirements grow, and as machines become available—that is, just-in-time. Furthermore, machines can also be reallocated to meet specified policies and constraints.

- *Support for diverse programming systems:* Supports a collection of unmodified parallel programming systems; ResourceBroker is the first resource manager that can simultaneously manage programs written in PVM, LAM [106] (an implementation of the MPI standard), Calypso, and PLinda [77] (an implementation of Linda with atomic transactions). Sequential and non-adaptive parallel programs are also supported.

- *System independence:* Parallel programming systems are treated as Commercial Off The Shelf (COTS) components. ResourceBroker does not require any modifications to programming systems and it works with existing compiled binary programs.

- *User privilege:* Super-user privileges are not required. The only privilege required for installation, configuration, and use are user-level access to managed machines. Thus, ResourceBroker does not compromise the security of the network.

Road Map

The rest of this chapter is organized as follows. Section 4.2 briefly describes the advantages of just-in-time allocation of resources, and the challenges that need to be addressed to provide such a service. The goals and the architecture of ResourceBroker are presented in Sections 4.3 and 4.4, respectively. Section 4.5 describes the specification language for conveying resource requirements to ResourceBroker, and ResourceBroker's techniques to communicate resource availability to programs. Section 4.6 describes the mechanism used to dynamically allocate, deallocate, and reallocate resources among running jobs. Experimental results are presented in Section 4.7. An overview of selected resource management systems, with an emphasis on resource allocation for adaptive jobs, is given in Chapter 7.

4.2 Background

On networks of machines that support both parallel jobs and interactive users, machine loads change over time. A number of studies [104, 127, 50, 101, 4, 39] indicate that in most institutions up to 60% of machines are idle at any given time. A machine is referred to as *idle* and, hence, *available* to participate in a computation when it is not used by its owner and its CPU cycles are mostly unused. The availability of machines is unpredictable because of users running remote jobs and machine owners re-claiming their machines for exclusive use.

The performance of parallel programs is very sensitive to machine loads and availability. Parallel programs with interdependent components cannot make progress if some of the participating machines become unavailable or heavily

loaded during a computation. In the experiments conducted by Arpaci *et al.* [4], when machines were shared by two parallel jobs, the jobs were slowed down by at least a factor of eight, and for some by a factor of 50. Thus, in environments where machines availabilities and their loads change over time, just-in-time allocation and reallocation of resources is essential in order to provide acceptable performance. The challenge is to provide the just-in-time allocation and reallocation of resources to programs written using multiple parallel programming systems.

Parallel programming systems such as PVM, MPI, Calypso, and Linda have a built-in resource manager that is of limited functionality. It is common for this type of resource manager to (1) schedule parallel tasks among the (already) allocated machines, and to (2) assist in adding explicitly-named machines to a computation. This type of resource manager is referred to as an *intra-job resource manager*, since its primary responsibility is to manage resources within one job. For example, PVM's default intra-job resource manager uses round-robin scheduling to assign PVM tasks to PVM daemons; and Calypso's intra-job resource manager employs Eager Scheduling to assign tasks to running worker processes. To differentiate from the resource manager built into parallel programming systems, *inter-job resource manager* is used to refer to the system that manages *all the machines among all the executing jobs*. The concept of separating these two types of managers was first discussed in the work on the Benevolent Bandit Laboratory [54].

Dynamic allocation and reallocation of machines to jobs requires communication between multiple inter-job resource managers and the intra-job resource manager. The lack of a common interface introduces a challenge to inter-job resource manager developers: how can their software system communicate with a variety

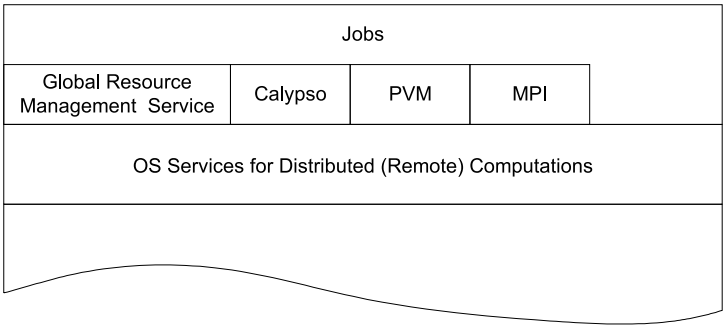


Figure 4.1: Structure of a global resource manager as a user-level service.

of parallel systems, especially with systems that do not provide an interface? The lack of a common interface is the reason existing resource managers for adaptive programs restrict themselves to supporting only a single programming system.

ResourceBroker is the first resource manager to use low-level features common to popular parallel programming systems for its communication. Hence, it is able to manage unmodified PVM, MPI, Calypso, and PLinda programs.

4.3 Design Goals

The goal of ResourceBroker is to provide a comprehensive resource management service that is able to dynamically allocate machines among multiple competing computations written in different parallel programming systems. The following is a list of the key objectives met in this system:

- It treats parallel programming systems as COTS. Unlike other resource managers, it does not expect programming systems to be modified or adapted to take advantage of its services. ResourceBroker can manage executables that

were compiled and linked for environments with no global resource management service.

- It treats operating systems as COTS. The current implementation runs on top of unmodified Linux 2.0.34. The mechanisms are general and are applicable to other Unix-type operating systems.

- The use of the resource manager is optional. To emphasize its unobtrusive availability, it is sometimes referred to as a "service." Figure 4.1 depicts this service in relation to programming systems and user programs. A user can choose to use ResourceBroker's services at each invocation of a job.

- The service can dynamically reallocate resources among adaptive jobs, as resource requirements grow and as machines become available.

The policy employed by resource managers in assigning resources to computations is a significant part of their service. Here the policy and the underlying mechanisms are separated, and focus of the ResourceBroker is on the mechanisms. But in order to test these mechanisms, the following policy has been implemented for ResourceBroker. User jobs are divided into two classes: adaptive, and others. Similarly, machines are divided into two classes: *private* and *public*. Private machines belong to individuals and the owner has absolute priority over its use, where as public machines are available to all users and typically reside in a public laboratory. The policy implemented by ResourceBroker is to allocate private machines only to adaptive jobs. Hence, adaptive jobs running on a privately owned machine can be deallocated once the owner of the machine returns. In other cases, ResourceBroker tries to "fairly" partition machines to running jobs.

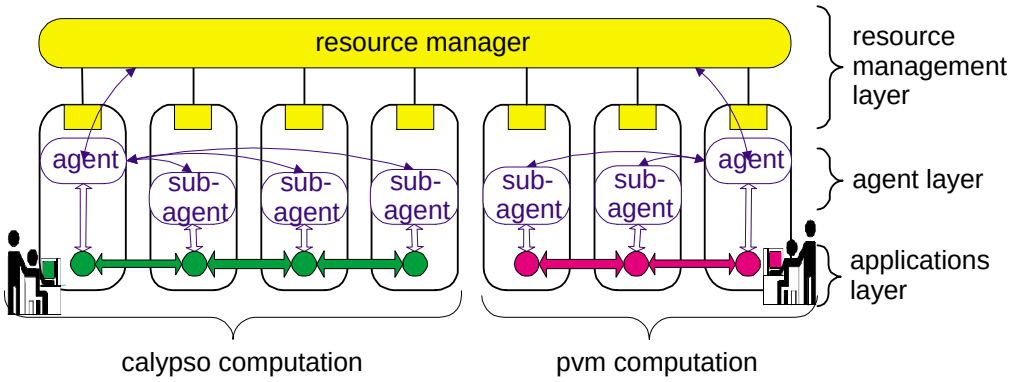


Figure 4.2: The components of ResourceBroker that comprise of the resource management and the agent layers

4.4 Architecture

ResourceBroker consists of two weakly coupled layers: the *resource-management layer* and the *agent layer*. Figure 4.2 depicts the software layers during the execution of two parallel jobs. Our research focuses on the mechanisms employed by the agent layer, but to validate and test the overall approach, the resource-management layer has also been implemented.

The *resource management layer* consists of a single network-wide resource manager process and a single daemon process on each machine. It is possible to run the resource manager process with non-privilege user access rights, as opposed to administrator access rights. The resource manager process spawns the daemon processes at startup and restarts them if they fail. Daemons are responsible for monitoring resources such as the CPU status, the users who are logged on, the number of running jobs, and the keyboard- and the mouse-status on the machine. This information is periodically reported to the resource manager process. The

resource manager process is responsible for deciding which jobs can use which machines.

The *agent layer* consists of dynamically changing sets of processes. A user who wants to use ResourceBroker's services first starts an *agent* process then submits the job for execution. As the job extends its execution to remote machines, *subagent* process are automatically started to monitor the new processes. This is illustrated in Figure 4.2. The combination of agent and subagent processes form the agent layer. The agent layer provides the means for the resource manager process to monitor and actively intervene in the execution of adaptive jobs. In a broader sense, it acts as a broker between the resource management layer and the running jobs, and is able to coerce programs to achieve the allocation policy determined by the resource management layer.

Many existing resource managers [92, 141, 103, 68] employ a single-level architecture, where the monitoring daemon processes also carry out the responsibilities of agent and subagent processes. The purpose behind a two-level architecture is to allow ResourceBroker to run with user-level privileges only. Although resource-management layer processes run with a user privilege, it is able to manage other user's jobs. These mechanisms are described in Section 4.6.

4.5 User, Job, and Resource Manager Interactions

In the presence of a resource manager, there are three types of entities in every job invocation: (1) the resource manager, (2) the user, and (3) the job submitted for execution. Figure 4.3 depicts the three entities and their interactions. The

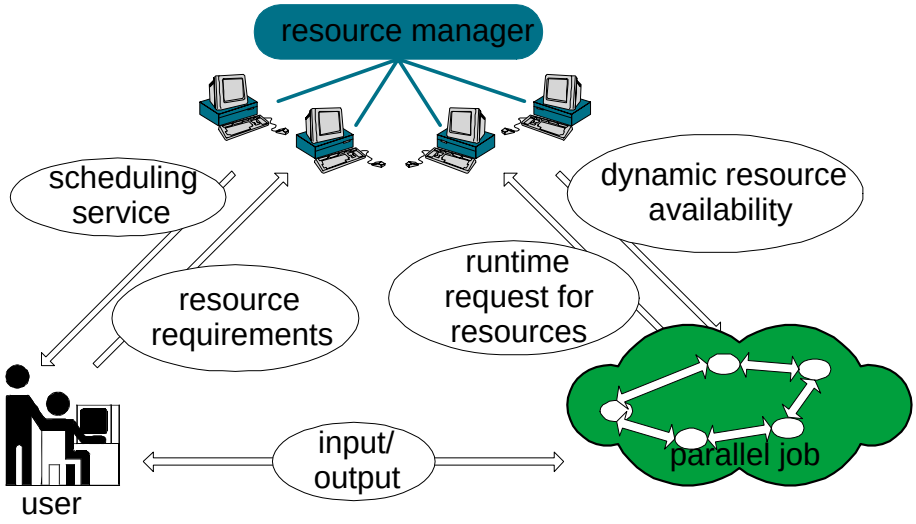


Figure 4.3: The three entities involved in every job execution, i.e., the resource manager, the user, and the job, and their interactions.

interaction between a user and the resource manager and the interactions between running jobs and the resource manager are of particular interest and are discussed next.

4.5.1 Users' Interaction with the ResourceBroker

Users communicate with the resource manager to query the availability of machines, to learn the status of queued jobs, to submit jobs for execution, and to specify a job's resource requirements. ResourceBroker adopted the Resource Specification Language of Globus [57], and extended it to support adaptive programs. Specifically, `adaptive`, `start_script`, and `module` parameters were added to describe adaptive jobs. The supported specification grammar is presented in Figure 4.4. As an example, the following specification

```
specification := request

request := multirequest | conjunction | disjunction | parameter

multirequest := "+" request-list

conjunction := "&" request-list

disjunction := "|" request-list

request-list := "(" request ")" request-list | "(" request ")"

parameter := parameter-name op value

op := "=" | ">" | "<" | ">=" | "<=" | "!="

parameter-name:= "adaptive" | "arch" | "count" |

  "max_time" |"module" | "start_script" | "start_time"

value := ([a..Z][0..9][_])+
```

Name	Default	Semantics
`adaptive`	false	whether the job is adaptive
`arch`	linux	execution architecture
`count`	1	number of machines to allocate
`max_time`	inf	maximum allocated time
`module`	NULL	(external) module name for communication
`start_script`	NULL	program to execute after allocation of resources and before invocation of the job
`start_time`	now	time to start the job

Figure 4.4: Specification language for describing job requirements.

```
+(count<=8)(adaptive=1)(start_script="my_script")
```

is a request to execute an adaptive program (as specified by `adaptive=1`) on up to eight machines. The `start_script="my_script"` requests the execution of `my_script` after the resources have been allocated, but before the job starts. The names of the allocated machines are passed as arguments to the start-scripts. A typical script to initialize a hostfile is the following:

```
#!  /bin/sh
echo add $* > $HOME/hostfile
```

PVM, LAM, and Calypso programs can use a hostfile to initialize the available pool of machines. As an additional example,

```
+(count>=4)(arch="i86Linux")(module="pvm")
```

is a request to execute a PVM program on at least four Intelx86 Linux machines (as specified by `arch="i86Linux"`). The module option (`module="pvm"`) specifies that ResourceBroker should use external modules to carry out some of its responsibilities. External modules are discussed in Section 4.5.2

4.5.2 Interaction of Jobs and ResourceBroker

Interactions between resource managers and jobs is complicated by the fact that these two software components are typically developed independently. To address this, ResourceBroker relies on a set of common features to build an interface between intra- and inter-job resource managers.

Communication from Job to ResourceBroker

Most of the parallel programming systems' intra-job resource managers are capable of relinquishing machines, but are unable to locate additional underutilized machines. As the requirements of adaptive jobs change over time, the resource manager must be alerted to these changes. Thus, at a minimum, adaptive jobs must inform the resource manager of their desire to add additional machines to a computation.

On Unix, the `rsh` command and the `rexec()` system call are the common underlying mechanisms to start program executions on remote machines, although the actual command available to the programmer is likely to be higher-level. For instance, Calypso and PVM computations grow by calling the `calypso_spawnWorker()` and the `pvm_addhosts()` library functions, respectively. In both cases, the function call results in a `rsh` command. This is also true if the computation pool is grown using the Calypso graphical user interface or using the PVM console.

The Unix `rsh` command requires an explicit machine name argument, as in "`rsh` *`host`* `<command>`."

ResourceBroker intercepts `rsh` commands issued by jobs running under its control. Intercepted `rsh` commands with *symbolic* host names, e.g. *anyHost*, are interpreted as intra-job resource managers' requests for assistance; `rsh` with real host names are allowed to proceed. This way, a job can inform ResourceBroker of its intention to add an additional machine by issuing "`rsh` *`anyHost`* `<command>`." Symbolic host names are also used as a request specification. For example, *anyLin-*

ux indicates any machine running Linux, and *anyLinuxMem128* indicates a Linux machine with 128Megs of RAM. As shown in Section 4.6, it is easy to make existing programs issue `rsh` commands with symbolic host names.

Communication from ResourceBroker to Job

Once ResourceBroker decides to allocate a machine to a running job, the action is carried out by the agent process (see Section 4.4) responsible for that job. Parallel programming systems that allow anonymous machines to join a computation are handle slightly differently from the systems that do not. The default behavior of the agent process is to replace the symbolic host-name with a real name and then to allow the `rsh` command to proceed—in a sense, the agent process redirects the `rsh` command to a machine unknown (anonymous) to the job. The default behavior is appropriate for Calypso, PLinda and sequential jobs. Some parallel programming systems, such as PVM and LAM, do not allow an unexpected (anonymous) machine to join a computation. In such cases, the agent process relies on external *modules* to communicate the real host name to the job and to coerce the job to accept it. Modules are executable programs, or shell scripts, that are external to ResourceBroker. This architecture allows future support for as yet undefined programming systems without having to recompile the resource manager.

When a user submits a job along with a module option, as in `module="xxx"`, ResourceBroker assumes the existence of three external programs named `xxx_grow`, `xxx_shrink`, and `xxx_halt`, to assist in growing, shrinking, and halting the job respectively. According to this naming scheme, PVM modules would be called `pvm_grow`, `pvm_shrink`, and `pvm_halt`. Figure 4.5 depicts the source code for PVM

```
#! /bin/bash
echo add $1 > $HOME/.pvmrc
echo quit >> $HOME/.pvmrc
pvm > /dev/null
rm $HOME/.pvmrc
```

(a) pvm_grow

```
#! /bin/bash
echo delete $1 > $HOME/.pvmrc
echo quit >> $HOME/.pvmrc
pvm > /dev/null
rm $HOME/.pvmrc
```

(b) pvm_shrink

```
#! /bin/bash
echo halt > $HOME/.pvmrc
pvm > /dev/null
rm $HOME/.pvmrc
```

(c) pvm_halt

Figure 4.5: PVM modules to grow, shrink and halt a PVM virtual machine.

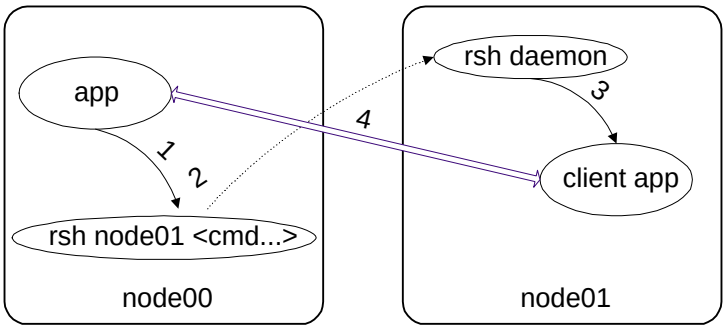


Figure 4.6: Representative scenario of how a parallel job acquires another machine.

modules. Consider `pvm_grow` for example: it writes a sequence of commands to `$/.pvmrc` and then invokes a PVM console to execute the commands. Notice how PVM modules are simple scripts that simulate users' actions; this is also true for LAM modules.

4.6 Mechanisms

To provide the context used in illustrating the mechanisms behind ResourceBroker, a representative scenario of how parallel jobs acquire resources *in the absence* of resource managers is briefly described. A user wants to run a program named `app` on machine `node00`, and wants the computation to grow to `node01` when needed. The user prepares a hostfile, named `.hosts`, containing `node01`. On `node00` the user types:

```
$app <arguments>.
```

The program starts executing. At some point `app` decides to spawn a process on another machine. Figure 4.6 depicts the steps involved in this process (note

the step numbers marked on the arrows). The `app` process consults `.hosts` for a machine name, reads `node01`, and issues the command "`rsh` *`node01`* `<command>`", as depicted in step 1 of Figure 4.6. In the job's source code this could have been a higher-level function, but ultimately it translated to the standard `rsh` command. The `rsh` command contacts the remote shell daemon (`rshd`) on `node01` (step 2), which spawns a process on `node01` on behalf of `app` (step 3). This new process establishes a communication stream with `app` (step 4). This completes the addition of `node01` to the computation.

When using ResourceBroker, selection of the second machine can be delayed until the job is ready to use the machine. It is the responsibility of the agent process to coerce the job into using a machine that is selected at runtime. Schematically, the agent process needs to accomplish the following tasks:

1. realize that the job "wants" to spawn a remote process
2. notify the resource manager that a new machine has been requested
3. obtain the name of the target machine that is "most appropriate"
4. spawn, or cause to be spawned, the second process on the target machine
5. enable the two processes to establish a communication stream
6. fade in to the background, so as not to impose an overhead

4.6.1 Required Conditions

ResourceBroker is capable of dynamically selecting resources for unmodified executables as long as the following requirements are met:

- The program does not contain hard-coded machine names. This is a natural assumption, because otherwise the program is not targeted for execution anywhere other than on a specific set of machines, which was defined at compile time.

- The program does not have the absolute path of the `rsh` command hard-coded. This is a natural assumption since in most cases either the user's default path is searched or, in the case of PVM and LAM, the location of `rsh` is specified during the installation of the programming system.

- For programming systems that do not allow anonymous machines joining a computation, there must be a command line interface for users to grow the pool of machines used in a computation. This is a common feature for many parallel programming systems, including PVM and LAM.

- For programming systems that do not allow anonymous machines joining a computation, it then must tolerate failed attempts to add additional machines. This is the case for PVM, LAM, Calypso and PLinda.

4.6.2 Default Behavior

Continuing with the previous example (page 90), to use ResourceBroker a user does two things. First, the user prepares the `.hosts` file containing *anyLinux*, a symbolic machine name. Second, on *host00* the user types:

```
$agent app <arguments>.
```

This starts an agent process, which immediately spawns a child process to execute `app`. Figure 4.7 depicts the steps involved for `app` to spawn a process on

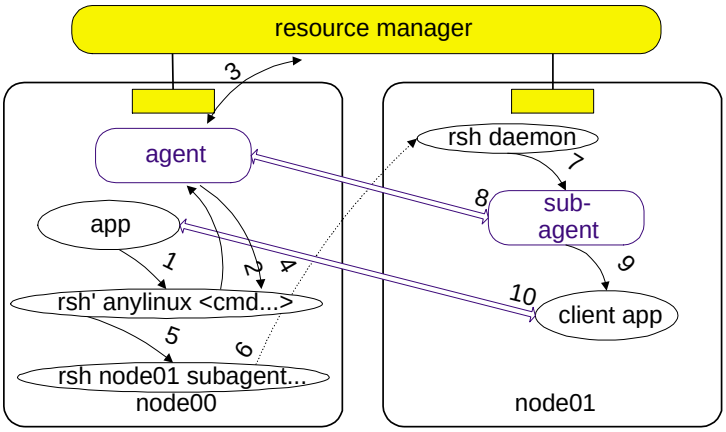


Figure 4.7: Adding a dynamically allocated machine (default behavior).

another machine when executing under ResourceBroker's control. When `app` decides to grow, it consults `.hosts`, finds *anyLinux*, and issues the command "`rsh` *anyLinux* `<arguments>`". See step 1 of Figure 4.7, where our implementation of `rsh` is depicted as `rsh`$'$. Realizing that *anyLinux* is a symbolic machine name, ResourceBroker intervenes. The `rsh`$'$ process contacts the agent (step 2), which contacts the resource manager process with a request for a machine (step 3). Once a target machine name is given to the agent, say `node01`, it notifies `rsh`$'$ of this machine name (step 4). Then `rsh`$'$ uses the standard `rsh` to spawn a subagent process on `node01` (steps 5-7). The subagent contacts the agent process for a program to execute (step 8), and spawns the appropriate process (step 9) on `node01`. The newly created process contacts the original `app` (step 10) and the job continues executing normally. Notice how the agent layer redirects the `rsh` to use a target machine selected at runtime.

From this point, until resources need to be reallocated, there is no interaction between `app` and ResourceBroker. The various agent-layer processes remain dor-

mant, and no overhead is imposed by their existence. Future interactions can begin in two ways: first, by the job attempting to add another machine; second, by the resource manager deciding to reallocate machines. To take away `node01` from `app`, the subagent sends a standard Unix signal to the child process, and if the child does not terminate within a specified amount of time, the subagent terminates the child process.

The default behavior described above is used for Calypso and PLinda programs, for parallelizable tasks such as `make`, and for executing sequential jobs remotely.

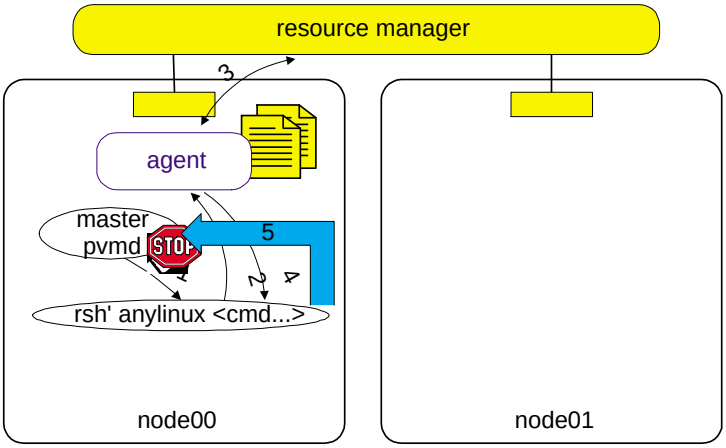
4.6.3 External Modules

In reviewing the default behavior, note that `app` running on `node00` attempted to spawn a process on a machine it believed to be *anyLinux*, whereas the process was spawned on `node01`. Generally, redirecting the `rsh` goes unnoticed. However, PVM and LAM programs will refuse to accept processes from machines other than those they attempted to spawn.

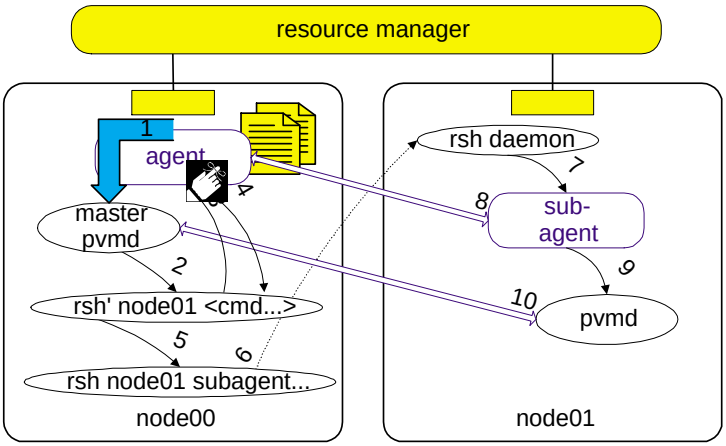
To handle this type of situation, ResourceBroker relies on external modules to carry out its responsibilities. A similar mechanism is used for both PVM and LAM programs: the "plug-in" external module approach makes the design extensible and thus able to accommodate various programming systems concurrently. A PVM-specific scenario, as illustrated below, is used as a representative usage of external modules.

Knowing that `app` is a PVM program, the user types:

```
$agent pvm --(module="pvm").
```



(a) Phase I



(b) Phase II

Figure 4.8: Adding a dynamically allocated machine (using external modules).

This submits the PVM console program, `pvm`, to an agent process and instructs ResourceBroker to use PVM-specific modules. The `agent` process immediately spawns `pvm`, which in turn starts the master PVM daemon. The user can create a PVM virtual machine and start PVM programs as usual.

PVM's virtual machine can grow in two ways: at the PVM console, the user can type

`pvm> add` *anyLinux*,

or the program can call the `pvm_addhosts()` library function with *anyLinux* as the host name argument. In both cases, this results in the master PVM daemon issuing "`rsh` *anyLinux* `<arguments>`". ResourceBroker intervenes when it detects an `rsh` with a symbolic host name. The allocation of resources using external modules happens in two phases. Figure 4.8 depicts the steps involved in this process.

Once the PVM daemon issues "`rsh` *anyLinux* `<arguments>`" (step 1 of Figure 4.8(a)), `rsh`$'$ contacts the agent (step 2), which asks the resource manager process for a machine (steps 2 and 3). The resource manager process knows that the job is a PVM task and propagates this information to `rsh`$'$ (steps 3 and 4). The `rsh`$'$ then terminates with an error status code (step 5). The first phase is completed with the result that (1) the master PVM daemon sees a failed attempt to grow the virtual machine, but more importantly, (2) ResourceBroker recognizes the request for an additional resource.

Figure 4.8(b) depicts the steps in the second phase of this process. Following ResourceBroker's recognition of PVM's request for an additional machine, phase two begins by the agent executing the external module `pvm_grow` with argument

host01 (step 1). This script consists of five lines and is shown in Figure 4.5. The script opens another PVM console, asks the master PVM daemon to add machine *host01* to the virtual machine, and closes the console. This results in the PVM daemon issuing another `rsh` command with `node01` as the machine name (step 2). The second phase proceeds like the default case and results in starting a PVM daemon process on *host01* (steps 3-10).

Three important features are worth emphasizing. First, the PVM daemon was coerced into accepting *host01*. Second, PVM's second attempt to add a host proceeds as usual; allocating "a suitable machine at the time of request" became an "invisible" service. Finally, as machines become available, ResourceBroker is able to asynchronously initiate the second phase to increase the size of PVM's virtual machine.

4.7 Experiments

This section presents experimental results that validate the proposed mechanisms. Experiments were conducted using up to 16 200MHz PentiumPro machines running Linux RedHat 4.0, that were connected by a Fast Ethernet hub. Reported times are median measured elapsed times taking into account all overheads. In this section, we use `rsh` to denote the standard Unix remote shell program, and `rsh'` to denote ResourceBroker's version.

Operation	Time (s)
`rsh` *n01* `null`	0.4
`rsh'` *n01* `null`	0.6
`rsh'` anyLinux `null`	0.6
`rsh` *n01* `loop`	36.9
`rsh'` *n01* `loop`	37.0
`rsh'` anyLinux `loop`	37.1

Table 4.1: Performance comparison of `rsh'` and `rsh`.

Micro Benchmarks

The first set of experiments compares the performance `rsh` and `rsh'`. The results are shown in Table 4.1. Two sequential applications were used for this experiment: `null`, a C program with an empty `main()` function; and `loop`, a C program with a tight loop that ran in 34.4 seconds. For this experiment, two idle machines, *n00* and *n01*, were used. The commands in Table 4.1 were issued on *n00* and directed to execute on *n01*. Thus, the command "`rsh'` ***n01*** `loop`" results in executing `loop` on *n01*. The *anyLinux* keyword is interpreted as "any available Linux machine." Thus, the command "`rsh'` ***anyLinux*** `loop`" allows the resource manager to choose a machine to execute `loop`. In this particular experiment, the available set of machines was limited to *n01*, so in fact *n01* was always chosen. As the results indicate, the overhead associated with `rsh'` is approximately 0.2 seconds, which is hardly noticeable by users. The small overhead also indicates that replacing the system-wide `rsh` with `rsh'` is feasible, even if some users do not use the features

Operation	Time (s)
`rsh` *n01* `null`	0.4
`rsh'` anyLinux `null`	1.5
`rsh` *n01* `loop`	38.2
`rsh'` anyLinux `loop`	37.9

Table 4.2: Performance of resource reallocation.

provided by ResourceBroker.

The second set of experiments measured the required times to reallocate machines. The results are shown in Table 4.2. Three machines were used in this experiment: *n00*, *n01*, and *n02*. An adaptive Calypso program ran on *n01* and *n02*. Similar to the previous experiment, the commands of Table 4.2 were issued on *n00*, and in every case resulted in the allocation of *n01*. In the case of `rsh'`, ResourceBroker terminated the Calypso process running on *n01* before satisfying the request. The results show that a reallocation completes in approximately 1 second. It is also interesting to note that in the case of `loop` (and other compute-intensive jobs), users experience a faster turnaround time since *n01* is cleared of external processes before executing the job.

Parallel Computations

This section presents the performance of ResourceBroker when managing parallel jobs. In particular, the effects of external modules within the agent layer are measured.

This experiment measured the performance of `rsh'` when used by parallel pro-

Operation	1 machine (s)	2 machines (s)	4 machines (s)	8 machines (s)
pvm w/ `rsh`	0.5	1.1	2.3	6.7
pvm w/ `rsh'` *host*	0.6	1.4	3.2	7.1
pvm w/ `rsh'` anyLinux	2.1	3.6	9.1	17.6
lam w/ `rsh`	1.8	1.8	2.1	2.0
lam w/ `rsh'` *host*	2.5	2.8	2.4	2.4
lam `rsh'` anyLinux	4.2	8.8	16.8	33.5

Table 4.3: Performance of ResourceBroker (using external modules) to dynamically allocate additional resources to PVM and LAM programs.

grams. The results are shown in Table 4.3. The operation "`pvm w/ rsh'` ***host***" means the PVM program explicitly named the machines it wanted to use, and "`pvm w/ rsh'` ***anyLinux***" means the PVM program left the choice of the machine to the resource manager. The results illustrate that when the machines are explicitly named, ResourceBroker introduces less than 0.1 milliseconds of overhead per machine. Allowing ResourceBroker to choose a machine (i.e, `rsh'` *anyLinux*) incurs approximately 1.4 seconds overhead for PVM and 3.5 seconds for LAM programs. This overhead occurs once per machine, and only at startup. This result reinforces the finding that replacing the system-wide `rsh` with `rsh'` is feasible, and that this will go unnoticed by users who do not use the additional features provided by `rsh'`.

The next experiment measures the time to reallocate resources for parallel jobs. The setting of the experiment was as follows. An adaptive Calypso job ran on every machine. A PVM virtual machine was created several times, and

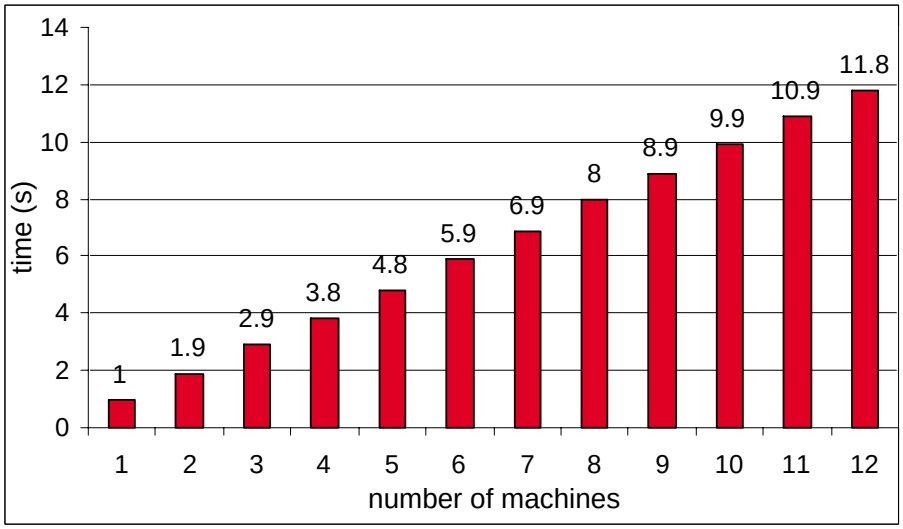


Figure 4.9: Performance of resource reallocation using PVM and `rsh`$'$.

each time a different size (denoted by n) virtual machine was built. To satisfy the PVM requests, machines had to be taken away from the Calypso job first. Figure 4.9 reports the elapsed times from the invocation until the resources were made available. The results show that the reallocation completes in approximately 1 second per machine (which is consistent with the second set of experiments), and that this number scales linearly to at least 12 machines.

The final experiment measures the utilization factor of a dynamic environment. The setting was as follows. An adaptive Calypso job initially ran on eight machines. Every 100 seconds, a script started a sequential program that ran for t minutes, where t was chosen uniformly from the interval [1,10]. After five hours the total detected idleness was less than 1%. This number indicates the efficiency of the reallocation mechanisms. Furthermore, it shows that in the presence of adaptive programs, a resource manager can boost utilization of a network to above 99%.

Chapter 5

Charlotte: Parallel Computing on the World Wide Web

5.1 Introduction

A recent report [102] indicates that the number of registered Internet Protocol (IP) [117] addresses has been growing at 50%–80% per year, in the last decade, and as of July 1998 there are over 36 million registered addresses. Other reports [104, 127, 50, 101, 4, 39] indicate that most networked machines in typical commercial organizations and universities are underutilized and mostly idle. Given this wealth of unused computing power, it is not surprising that several projects [46, 47, 116, 115, 114, 138] have used the Internet to execute programs that are too compute-intensive even for a supercomputer. For example, in discovering the world's largest known prime number [136], PrimeNet reported a sustained throughput of more than 200 billion floating point instructions per second, the equivalent of seven

fully-equipped Cray T916 supercomputers at peak performance [137]. In another case, during the first successful brute force "crack" of a DES-encrypted message, it was observed that within a single 24-hour period nearly 14,000 machines joined the computation [42]; this is a tremendous amount of computing power.

The difficulties encountered by these projects arose from developing and deploying programs. For example, Curtin *et al.* [42] reported that because those who wanted to volunteer had to download and install a client software, "know what it did, and usually had to put forth some effort to keep it running," this limited the number of machines that joined the computation. Furthermore, 40 different versions of the client program—for a variety of hardware and operating system—had to be implemented to ensure that the most popular platforms could join the computation. In addition, the programs had to implement load balancing and fault masking. Clearly, a software system that resolves these difficulties could contribute to a proliferation of Web-based computing.

There are many software systems [141, 32, 35, 45, 10, 112] for developing and executing programs using idle CPU cycles in networks of workstations. Based on their success and as a natural evolutionary step, attempts have been made to extend such systems to the Web.

However, the challenges involved in utilizing the Web as a parallel processing resource are different from the challenges involved in utilizing networks of workstations, and many of the assumptions made for networks of workstations are not valid for the Web. For example, the machines on the Web do not have a common shared file system, no single individual has access-rights (user-account) on every machine, and the machines are not homogeneous. Another important distinction is the con-

cept of *users*. On networks of workstations, there are two entities (individuals or groups of individuals) involved in developing and executing programs: *programmers* who develop programs, and *users* who execute these programs. To execute a parallel program typically the user first logs onto a machine under her control (i.e. the *local* machine), then from the local machine logs onto other machines on the network (i.e. *remote* machines) and initializes the execution environment, and then starts the program. In the case of the Web, no user can possibly hope to have the ability to log onto remote machines. Thus, another set of users who control remote machines, or software agents acting on their behalf, must voluntarily allow others access. To distinguish the two types of users, this chapter uses the term *end-users* to refer to individuals who start the execution (on their local machines) and await results, and *volunteers* to refer to individuals who voluntarily run parts of end-users' programs on their machines (remote to end-users). Similarly, *volunteer machines* is used to refer to machines owned by volunteers.

Challenges

An end-to-end solution for metacomputing on the Web must address the concerns of programmers, end-users, and volunteers.

Programmers: As with other programs, the ease with which Web-based programs are developed and maintained is the main concern of programmers. Programmers prefer to use a high-level programming language with simple syntax and well-understood semantics. Programmer usually prefer to avoid the complexities associated with the development of programs for failure-prone and unpredictable

environments. The Web is a highly failure-prone and unpredictable environment—much more so than local area networks. I conjecture that if the Web is to become a practical metacomputing environment, the programming model must be decoupled from the execution environment. Then programs can be developed for a reliable virtual machine—thus isolating programmers from the dynamics of the Web—and execute on a pool of unreliable machines. Furthermore, the virtual machine concept facilitates the development of portable and heterogeneous programs—two important and time consuming issues for programmers.

End-users: End-users are mostly concerned with the ease, correctness, and efficiency of program executions. On local area networks, end-users can log onto multiple machines to execute a parallel program. This is made possible by having a single administrative control (to allow remote logins), and the existence of a shared file system (to make the program available on all machines). This is not a possible scenario for the Web, but it is desirable to have a software system that allows program executions on the Web to be as simple as local area networks. Finally, end-users need assurance that parts of their program executing on untrusted machines will not harm the accuracy of the computation. Validating the work performed on untrusted machines has been addressed in [99].

Volunteers: Simplicity and security are important objectives for volunteers. Unless the process of volunteering a machine is simple—for example as simple as a single mouse-click—and the process of withdrawing a machine is simple, it is likely that many would-be volunteer machines will be left idle. Furthermore, volunteers

need assurance that the integrity of their computer and file system will not be compromised by allowing "strangers" to execute computations on their machines. Without such an assurance, it is natural to assume security concerns will outweigh the charitable willingness volunteering.

The combination of the Java programming language and Java-capable browsers has successfully addressed some of the difficulties of Web-based computing. Java's platform independence solves the problem of heterogeneity. The growing number of browsers able to seamlessly load applets from remote sites reduces administration difficulties. The applet security model, which in most parts enables Web browsers to execute untrusted applets in a controlled environment, alleviates some of the volunteers' security concerns. These factors make the combination of Java applets and Java-capable browsers a good starting point to seamlessly bring distributed computing to every-day users. *Charlotte* builds on these advances by providing a comprehensive programming system.

Contributions

A comprehensive solution to facilitate Web-based computing involves assisting programmers, end-users and volunteers. We have designed and implemented two systems: *Charlotte* [15] and *KnittingFactory* [13], which in unison provide such a solution. *Charlotte* is an implementation of the metacomputer presented in Chapter 2 and is the focus of the present chapter; *KnittingFactory* is discussed in Chapter 6. The work on *Charlotte* has resulted in several original contributions, which are summarized below:

- *Charlotte* is the *first parallel programming system to provide one-click computing.* The idea behind one click computing is to allow volunteers from anywhere on the Web, and without any administrative effort, to participate in ongoing computations by simply directing a standard Java-capable browser to a Web site. A key ingredient in one-click computing is its lack of requirements: user-accounts are not required, the availability of the program on a volunteer's machine is not assumed, and system-administration is not required.

- *Charlotte* is the *first system for parallel computing that uses a secure language and execute in a secure sandbox environment.* It is implemented entirely in Java without any native (non-Java binary) code. Therefore, volunteers have the same level of trust in running *Charlotte* programs as they do in running any other Java applet.

- Existing contributions are leveraged in providing a *Virtual Metacomputer on the Web*. The programming environment is conceptually divided into a virtual machine model and a runtime system. The virtual machine model, as presented to the programmer, provides a reliable shared memory machine. The runtime system implements this model on a changing set of unpredictable machines.

- Our previous work developed for networks of workstations is extended to deal with the dynamics of the Web. Three integrated techniques—*eager scheduling*, *two-phase idempotent execution strategy*, and *bunching*—are used for load balancing, fault masking, and efficient execution of fine-grain tasks.

Road Map

The rest of this chapter is organized as follows. Section 5.2 provides the syntax and semantics of *Charlotte* programs. The steps involved in executing *Charlotte* programs are presented in Section 5.3. *Charlotte* implements some of the mechanisms described in Chapter 2, specifically, eager-scheduling, TIES, and bunching. The *Charlotte*-specific implementation issues of these mechanisms are described in Section 5.4. Performance results are presented in Section 5.5. Section 7.3 compares *Charlotte* with other related systems.

5.2 How to Write Charlotte Programs

A *Charlotte* program is written by inserting any number of *parallel steps* onto a sequential Java program referred to as *sequential steps*. A parallel step is composed of one or more *routines*, which must complete for the program to proceed to the next step. A routine is a (sequential) thread of control capable of executing on remote machines. Figure 5.1 shows an example of a *Charlotte* computation with two parallel steps. *Charlotte*'s programming model is sometimes referred to as a block-structured fork/join model [44, 107]. As previously mentioned, this programming model along with shared memory semantics allows loop-level parallelization. Thus, given a working sequential Java program it is fairly straightforward to parallelize individual loops in an incremental fashion.

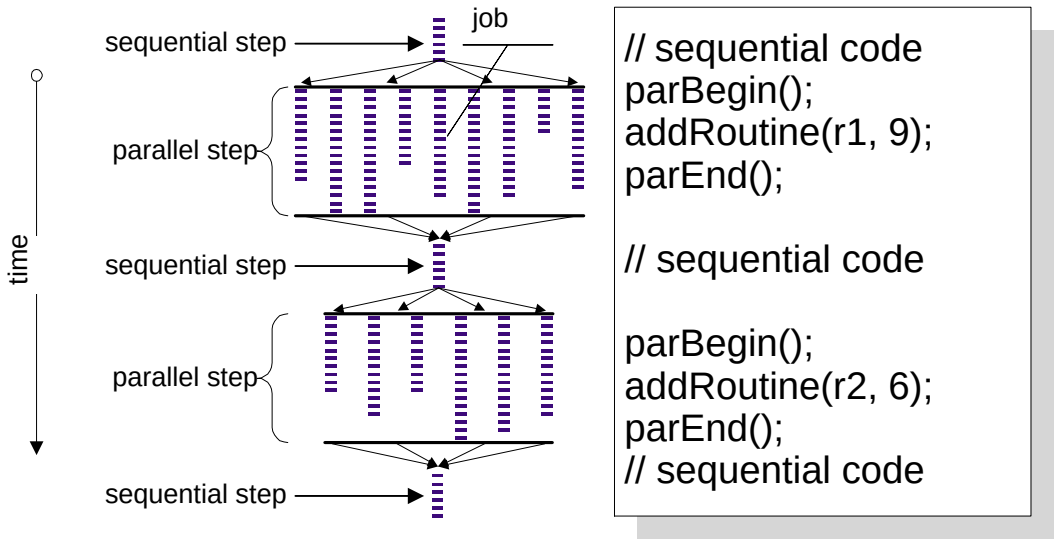


Figure 5.1: An execution of a *Charlotte* program with two parallel steps and three sequential steps; the first parallel step consists of 9 routines, the second parallel step consists of 6 routines.

Parallel Steps

A parallel step starts and ends with the invocation of `parBegin()` and `parEnd()` methods, respectively. A routine is written by subclassing the `Droutine` class and overriding its `drum()` method. Within a parallel step, routines are specified by invoking the `addRoutine()` method with two arguments: a routine object and an integer, n, representing the number of routine instances to execute. To execute a routine, the *Charlotte* runtime system invokes the `drun()` method of routine objects, and passes as arguments the number of routine instances created (i.e. n) and an identifier in the range $(0, \cdots, n]$ representing the current instance.

Distributed Shared Class Types

Charlotte programs are written for a shared memory virtual machine and they execute on the Web—obviously a shared-nothing environment. A program's data is logically partitioned into *private* and *shared* segments. Private data is local to a routine and is not visible to other routines; shared data, which consists of *shared class-type* objects, is distributed and is visible to all routines. The runtime system maintains the coherence of shared data.

For every basic data-type defined in Java, *Charlotte* implements a corresponding distributed shared class-type. For example, Java provides `int` and `float` data-types, whereas *Charlotte* provides `Dint` and `Dfloat` classes. The class-types are implemented as standard Java classes, and are instantiated by invoking their corresponding `newInstance()` method. Shared class-types are read and written by invoking `get()` and `set()` method calls, respectively.

Shared class-types must be instantiated in the same order at each site. The rationale behind this requirement is that the system needs to form a one-to-one mapping between class-type object instances at the various sites; instantiating them in the same order allows each object to be assigned the same identifier on all sites. This requirement is typically met by instantiating all class-type objects in the constructor of a single class.

Shared Memory Programming Model

Charlotte leverages its structured fork-join programming model in providing an intuitive shared memory semantics: *Concurrent Read, Concurrent Write Common* (CRCW-Common). This means that within a parallel step one or more routines can read the value of a variable, and one or more routines can write the same value to a variable. Read operations of unmodified data return the value of variables at the time the parallel step began; results of write operations become visible at the completion of the parallel step. The advantage of *Charlotte*'s memory model is that, semantically, routines execute in isolation, and hence the final result is independent of the execution order.

Example Program: Parallel Matrix Multiply

Figure 5.2 shows the relevant fragments of a program to multiply two 500×500 matrices in parallel; there are 500 routines, each responsible for computing a single row of the resultant matrix. This simple example illustrates the following important points:

- The parallelism reflects the parallelism inherent in the algorithm and not the

```
import charlotte.*; // import the Charlotte class files

public class MatrixMult extends Droutine {
    public static int  Size = 500;
    public Dfloat a[][] = new Dfloat[Size][Size];
    public Dfloat b[][] = new Dfloat[Size][Size];
    public Dfloat c[][] = new Dfloat[Size][Size];

    public MatrixMult() {
        // instantiate a, b and c
    }

    public void drun(int numTasks, int id) {
        int   sum;
        for(int i=0; i<Size; i++)  {
            sum = 0;
            for(int j=0; j<Size; j++)
                sum += a[id][j].get()*b[j][i].get();
            c[id][i].set(sum);
        }
    }

    public void run() {
        // initialize elements of a and b
        parBegin();
        addDroutine(this, Size);
        parEnd();
    }
}
```

Figure 5.2: Matrix multiplication in *Charlotte*.

execution environment. The same program can execute on one or any number of machines.

- The programmer need not be aware of the fact that the program executes on a distributed platform.
- Integrating machines onto the computation, load balancing, fault masking, and data coherence are transparent in the program.
- The *Charlotte* API does not require any language or compiler modifications.

Charlotte is implemented as a Java package without any native code. Hence, *Charlotte* programs are compiled as standard Java programs.

5.3 How to Run Charlotte Programs

To run a *Charlotte* program, the end-user must first start running the program on her local machine, then volunteers can join the running program from anywhere on the Internet. The end-user's machine is required to be non-faulty; volunteer machines can join and leave a running program, and crash at any time. As long as at least one volunteer machine does not continuously crash, programs will run to completion.

To start a program called `RayTrace`, the end-user types:

```
$java RayTrace <arguments>.
```

The program starts as a single process called the *manager*, which executes the sequential steps of the program. At startup, a manager produces an output similar to the following:

host = sunra.milan.cs.nyu.edu/128.122.142.137
port = 1096
Accepting computation requests at http://sunra.milan.cs.nyu.edu:1097/index.html
Accepting status requests at http://sunra.milan.cs.nyu.edu:1097/status.html
Registered with http://milan.milan.cs.nyu.edu/knittingfactory 5

The first two lines of the output contain the name, the IP address of the machine executing the manager, and its port number; the next two lines contain the URL [21] addresses where volunteer computations are accepted and status reports are served (this service is provided by *KnittingFactory* class servers, which are described in Section 6.4); the last line contains the address of the *KnittingFactory* directory server where the program is registered (*KnittingFactory* directory services are described in Section 6.3).

The parallel routines are executed by *worker* processes running on volunteer machines. Worker processes can start in three ways. The first method is to run a worker as a Java application. This is done by typing the following:

```
$java charlotte.Cdaemon <host> <port>.
```

Where the `<host>` and `<port>` are the host name and the port number of the manager. In the above example, the host and the port number are `sunra.milan.cs.nyu.edu` and `1096`, respectively. A more flexible method is to run a worker as a Java applet. This is simply done by directing a Java-capable Web browser to the manager's URL where computations are accepted. For example, the following command starts a Web browser, loads the code for the *Charlotte* worker process, and starts executing it:

`$netscape http://sunra.milan.cs.nyu.edu:1097/index.html`.

By running a worker as an applet, the program code is transmitted to the volunteers' machines, thus alleviating the need for having the code local to each machine. Furthermore, since applets run in a secure sand-box environment, volunteers can safely execute untrusted code. There is an implicit assumption that volunteers "know" *a priori* the URL address of *Charlotte* managers. This is a reasonable assumption for small networks, but not for the Web. The third method of starting a worker relies on a directory service to overcome this limitation. As previously indicated, managers can register with a *KnittingFactory* directory server during initialization. Once this happens, a volunteer can direct a Java- and JavaScript-capable browser to any *KnittingFactory* directory server to find a running program.

5.4 Implementation

Worker Process

A *Charlotte* worker process is implemented by the `Cdaemon` class which can run either as a Java application or as a Java applet. It requires two arguments, namely the host and the port of a *Charlotte* manager process to initialize. At instantiation, a `Cdaemon` object establishes a TCP/IP connection to the manager and maintains this connection throughout the computation. A worker process repeatedly performs the following sequence of tasks:

1. invalidates shared class-type objects,

2. contacts the manager for work,

3. receives a set of routines to execute,

4. downloads and instantiates the routine objects, if it hasn't done so already,

5. executes each routine in turn by calling its `drun()` method, and

6. sends the modified values to class-type objects back to the manager.

Two implementation features are worth noting. First, since `Cdaemon` is implemented as an applet (as well as an application), the code does not need to be present on volunteer machines before the computation starts. By simply embedding the `Cdaemon` applet in an HTML page, browsers can download and execute the worker code. Second, the `Cdaemon` class, unlike its counterpart the Calypso worker, is independent of the *Charlotte* program it executes. Thus, not only are *Charlotte* workers able to execute parallel routines of any *Charlotte* program, but only the necessary code segments are transfered to volunteer machines.

Manager Process

A manager process begins with the `main()` method of a program and executes the non-parallel steps in a sequential fashion. It also manages the progress of parallel steps by providing scheduling and memory services to workers.

The manager's scheduling service is responsible for assigning routine(s) to volunteer workers. At the beginning of each parallel step, the scheduling service populates *DispatchTable* from the `addRoutine()` method calls. The DispatchTable records which routines need to be assigned, which routines have been completed, and which routines have been assigned but not yet completed. As workers request

work from the manager, the routines that have not been assigned to any worker are scheduled first. If, however, some of the routines have been assigned but have not yet completed, the scheduler will aggressively re-assign the routine, or one of the routines, that has been assigned the least number of times—this technique is referred to as *eager scheduling* and is described in Section 2.2.1. Eager scheduling is able to mask failed worker processes resulting from crashed machines, networks, and volunteers stopping the process. Furthermore, because fast machines can re-execute routines that were initially assigned to slow machines, a slow volunteer machine will not slow down the progress of the program.

As mentioned in Section 2.2.3 *Charlotte* employs *bunching* to execute fine-grain parallel tasks in a coarse-grain manner. The scheduling service assigns a group of routines to workers by specifying a range of `id` values. The worker executes each routine in turn, sending the shared data modifications to the manager before executing the next routine. Bunching not only reduces the number of times routines are assigned to workers, but it overlaps computation with communication by allowing workers to execute routines while the shared memory modifications are sent back to the manager. The performance improvements due to bunching are reported in Section 5.5, which describes the results of running the same computation with and without bunching.

Distributed Shared Class Types

While executing a parallel routine, a worker process must page-in the values of shared objects necessary for its execution. Similarly, it must return the modifications to shared objects at the end of its execution.

Charlotte's distributed shared memory is implemented in pure Java at the data-type level; that is, through Java classes as stated above. For each primitive Java type like `int` and `float`, there is a corresponding *Charlotte* class-type `Dint` and `Dfloat`. The member variables of these classes are a `value` field of the corresponding primitive type, and a `state` flag that can be `not_valid`, `readable`, or `dirty`. A `not_valid` state indicates that the object does not contain a correct value; `readable` indicates that the object contains a correct value; and `dirty` indicates that the local value is correct and that it has been modified.

A read operation on a `readable` or a `dirty` object returns its `value`. A read operation on a `not_valid` object causes a "page-fault," that is, its value is retrieved from the manager and it changes state to `readable` before its value is returned. To reduce the number of page-faults, a set of several values is retrieved on each page-fault. Retrieving a set of values roughly corresponds to retrieving a "page" of data, but the size of the page can dynamically change to accommodate the program. A write operation on a class-type object modifies the `value` field of the object and changes its state to `dirty`. A `dirty` object propagates its modified value back to the manager at the completion of the routine. There is a cost associated with collecting the set of modified class-type objects at the end of each routine. For Calypso as well as other standard distributed shared memory systems, a large portion of this cost is caused by producing the page-level differences (or diffs) needed to send back to the manager [140]. In *Charlotte* this approach would correspond to scanning all (of the dirty pages) of the shared region for modified data. *Charlotte*'s implementation, however, reduces this cost by maintaining a set of dirty objects. This implementation is possible because shared memory is provided at the object

level. Section 5.5 compares the performance of both implementations.

Recall that *Charlotte* implements *two-phase idempotent execution strategy*, which specifies that shared memory modifications may not be visible until the parallel step completes. This is implemented at the manager side by buffering the modifications until every routine (in the current parallel step) has been executed at least once, at which point the modifications are applied to the manager's local memory. In addition, the implementation ensures that late updates (i.e. modifications for a routine that has already been completed) are discarded.

It is important to note that different parts of the shared data can be updated by different worker processes without *false-sharing*, as long as the CRCW-Common condition is met. The shared memory is always logically coherent, independently of the order in which routines are executed.

5.5 Experiments

A computational physics and a graphical application were used to evaluate the performance of *Charlotte*. Experiments were conducted on a set of machines under different patterns of slowdowns, failures, and recoveries.

Setup

All experiments, unless noted, were conducted on up to 17 identical 200 MHz PentiumPro machines running Linux version 2.0.34 operating system, and connected by a 100Mbps Ethernet through a non-switched hub. The network was isolated to eliminate outside effects.

Programs were compiled and executed in the Java Virtual Machine (JVM) packaged with Linux JDK 1.1.5 v7. TYA version 0.07 [82] provided just-in-time compilation. All programs were compiled with optimization. A C program would run faster than a Java program; however, as Java compilers and virtual machines continue to improve, they will provide better performance and it is expected that the results will carry over transparently.

Reported times are "wall clock" elapsed times, and not CPU or virtual times. It should be stressed that at the beginning of the measurements, volunteers did not have the shared data, and at the end of the measurements, the manager had received and processed the modified data. Thus, overheads associated with networking, swapping, and updating shared data regions are included in the measurements.

In order to experiment with heterogeneous machine speeds, several machine *profiles* were defined. A profile determines a machine's behavior. Below are the descriptions of each machine profile, see also Figure 5.3.

- *Machine* A_{100} makes 100% of its CPU available to the computation. This is a machine that does not fail or slow down during the execution. A machine with profile A_{100} models a perfect fast machine.

- Similarly, *Machines* A_{75}, A_{50}, *and* A_{25}, contribute to the computation with only 75%, 50%, and 25% of their CPU cycles, respectively. The slowdowns were achieved by running a (parameterized) high priority background process called the *hog*. The slowdowns were verified by ensuring that the execution of a standard benchmark took 133%, 200%, and 400%, respectively, longer

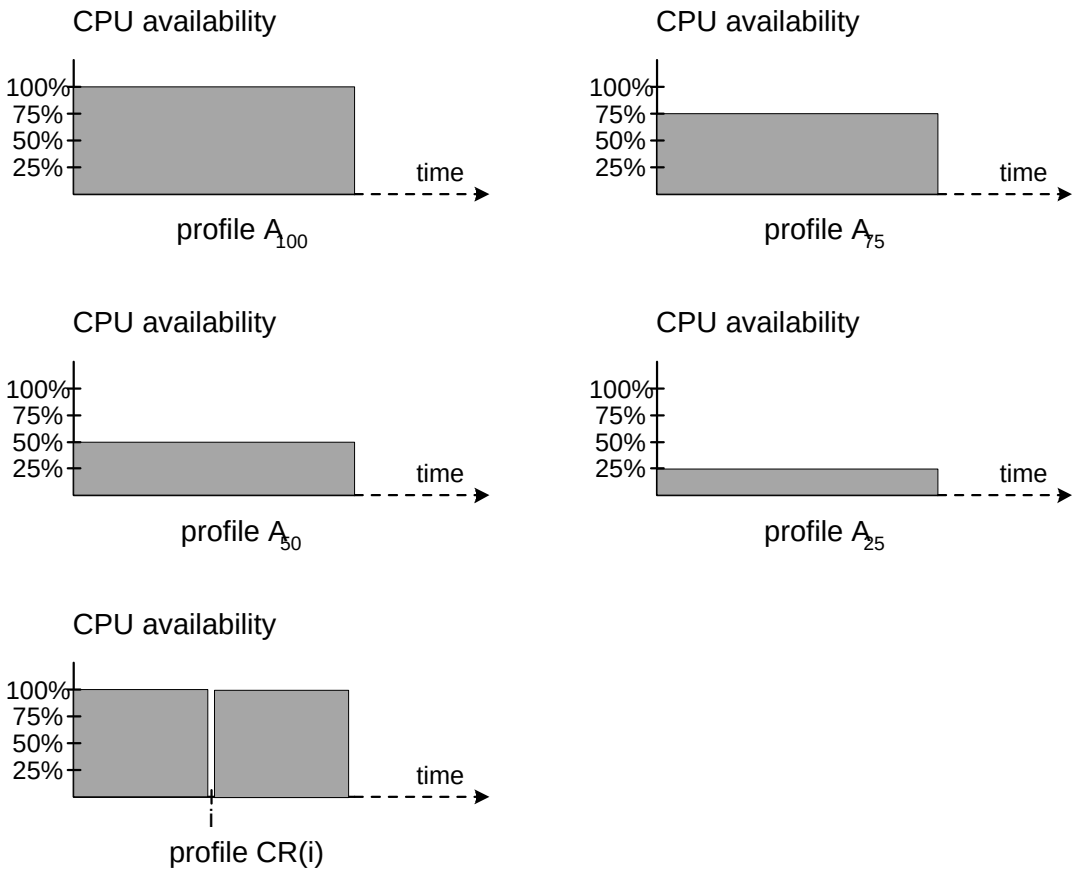


Figure 5.3: Profiles of volunteer machines.

to execute with the hog running. Machines with profiles A_{75}, A_{50}, and A_{25} model slower machines.

- *Machine* $CR(i)$ crash-fails after i seconds from the start of the computation and immediately recovers. This is achieved by manually killing the volunteer process at the corresponding time and then starting a new process. Experiments with machines of profile $CR(i)$ illustrate how well failures are masked and how efficiently a new volunteer can be incorporated into an ongoing computation.

Ising Model

Ising is a scientific application from statistical physics, which computes a 3D Ising model [24]. This is a simplified model of magnets on a three dimensional lattice and is used to qualitatively describe the behavior of small systems. Computing the Ising model involves a large number of independent tasks and little data movement.

A sequential program to compute the Ising model with a period of 23 was implemented in 288 lines of code. The program executes in 440.5 s; this number was used as the base case for computing speedups and efficiencies of a parallel implementation. A *Charlotte* program for exactly the same problem was implemented in 302 lines of code. The program consisted of a sequential initializing step, followed by a parallel step with 1023 routines, and a closing sequential step to collate the results. The same *Charlotte* program was used in each case and the runtime parameters did not change. The manager ran on a A_{100} machine and various profiles were used for volunteer machines. Four series of experiments were conducted as

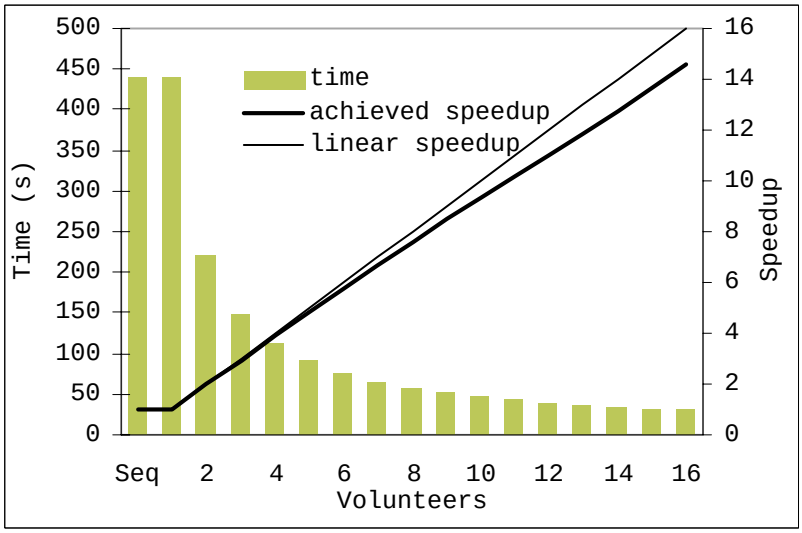


Figure 5.4: Scalability experiment of a *Charlotte* Ising model program.

described below.

The first series of experiments illustrates the speedup and scalability of *Charlotte*. The execution times of the same program, with the same runtime parameters, using a range of 1 to 16 A_{100} volunteer machines are reported in Figure 5.4. The speedup ranged from 0.99 for one volunteer to 14.55 for 16 volunteers. These numbers are gratifying, given such a high-level programming model and the lack of low-level optimizations.

The next set of experiments illustrates the importance of two optimization techniques: bunching, and avoiding the scan to collect modified class-type objects. Figure 5.5 compares the efficiency of the *Charlotte* program (the same as the previous series), with a modified version that does not perform bunching, and with another modified version that scans the entire shared memory region to collect modified objects. The results indicate that bunching and maintaining a list of

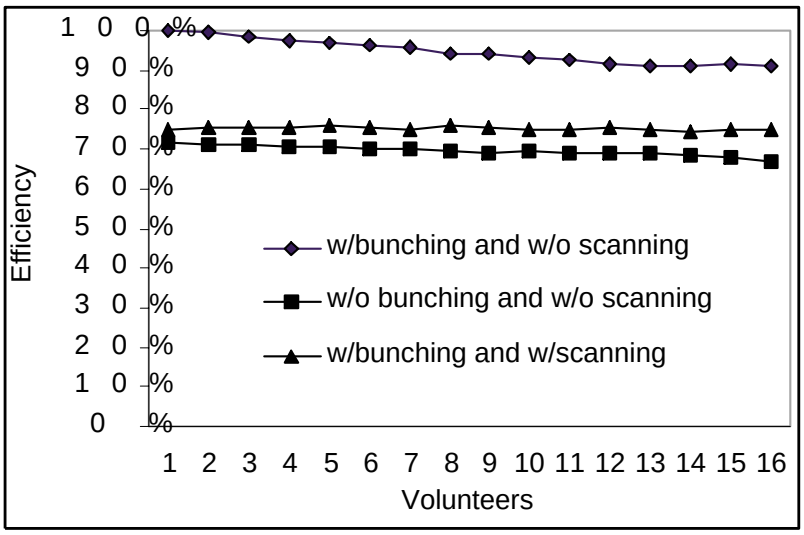


Figure 5.5: Effects of bunching for executing fine-grain tasks.

modified objects improve the efficiency of the computation by approximately 27% and 20%, respectively.

The third series of experiments illustrates load-balancing capabilities of *Charlotte*. The execution times of the same *Charlotte* program with the same runtime arguments, but with volunteer machines of varied speeds were measured. Each test had some α A_{100} machines and β A_{50} machines. The constraint $1.0\alpha + 0.5\beta = 8$ was observed in all experiments, thus assuming "volunteering potential" equivalent to eight fast machines. In fact the number of machines ranged from 8 to 16. See Figure 5.6 where a label of $\alpha A100 + \beta A50$ represents α A_{100} machines and β A_{50} machines. The speedup ranged from 7.26 to 7.53 using (the equivalent of) eight volunteer machines. The results show that load balancing has a low overhead.

The final series of experiments illustrates *Charlotte*'s ability to mask crashed volunteers and to incorporate new volunteers into computations. The following

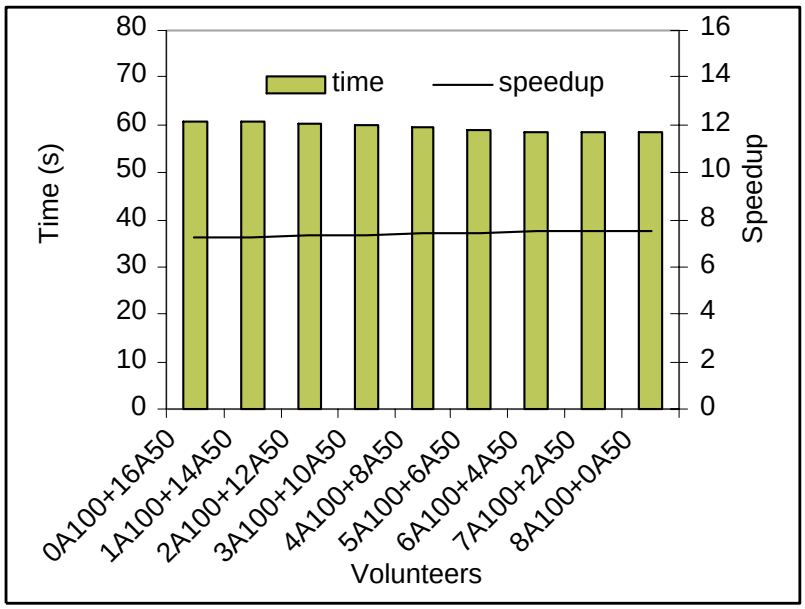


Figure 5.6: Load balancing of a *Charlotte* Ising model program.

five experiments started with four volunteer machines, but some of the volunteer machines were of type $CR(i)$. Such a volunteer machine ran for i seconds, then its JVM crashed, and then came up as a new volunteer. (This combined experiment was conducted in the interest of concise presentation.) Specifically, in the reference experiment there were four machines with profile A_{100}; in the first, there was one type $CR(20)$ machine and three type A_{100} machines; in the second, there was one type $CR(20)$ machine, one type $CR(40)$ and two type A_{100} machines; in the third, there was one type $CR(20)$ machine, one type $CR(40)$, one type $CR(60)$, and one type A_{100} machine; in the fourth, there was one type $CR(20)$ machine, one type $CR(40)$, one type $CR(60)$, and one type $CR(80)$. The speedups ranged from 3.90 (the reference experiment) to 3.52 (the fourth experiment). This is a satisfying

	number of files	lines of code (LOC)	relative increase of LOC
sequential	1	476	N/A
Charlotte	1	538	13%
RMI	3	627	31%
JPVM	3	689	45%

Table 5.1: Comparison of sequential, *Charlotte*, RMI, and JPVM ray tracing programs.

result, as the program managed the speedup of 3.52 while coping with four crashes and integrating four volunteers during the computation.

Ray Tracing

A publicly available sequential ray tracing program [98] was used as the starting point to implement parallel versions in *Charlotte*, Java RMI [52], and JPVM [55]. Java RMI is an integral part of Java 1.1 standard and, therefore, it is a natural choice for comparison. JPVM is a Java implementation of one of the most widely used parallel programming systems called PVM [59].

The *Charlotte* program was parallelized at the row-level, so each routine computed one row of the final image. The RMI and the JPVM programs were implemented using the manager-worker programming model (for basic load-balancing) and a runtime argument determined the granularity of parallel tasks. Table 5.1 compares the lines of code and the number of files used for the four different implementations. As depicted, the *Charlotte* program is the shortest of the three parallel implementations. An important difference (that is hard to measure) is the structure of the programs: in implementing the *Charlotte* program the structure of the

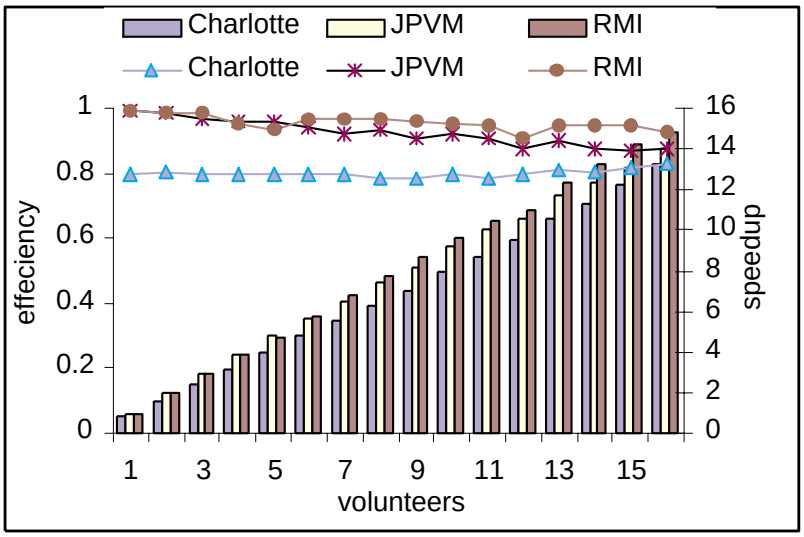


Figure 5.7: Performance comparison of *Charlotte*, RMI and JPVM programs.

sequential program was not changed and the modifications were localized; in contrast, the RMI and JPVM implementations required restructuring the sequential program and decomposing it into several files.

For the experiments a 500×500 image was traced. The sequential program took 154 s to complete, and this number is used as the base in calculating the speedups.

The first series of experiments compares the performance of the three parallel implementations of RayTrace, see Figure 5.7. In the case of *Charlotte*, the same program with the same runtime arguments was used for every run—the program tuned itself to the execution environment. For RMI and JPVM programs, on the other hand, executions with different grain sizes were timed and the best results are reported—the programs were hand-tuned for the execution environment. The results indicate that when using 16 volunteers, the *Charlotte* implementation runs

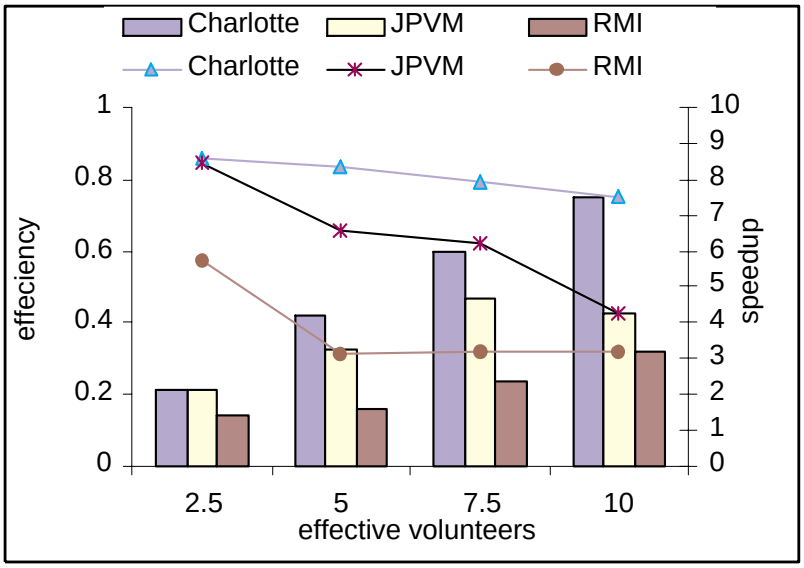


Figure 5.8: Load balancing of *Charlotte*, RMI and JPVM programs.

within 5% and 10% of hand-tuned JPVM and RMI implementations, respectively. It is encouraging to see that the performance of *Charlotte* is competitive with other systems that do not provide load balancing and fault masking.

The final set of experiments illustrates the efficiency of the programs when executing on machines of varying speeds—a common scenario when executing programs on the Web. Exactly the same programs with the same granularity sizes as the previous experiment were run on n, $1 \leq n \leq 4$, groups of volunteers, where each group consisted of one A_{100}, one A_{75}, one A_{50}, and one A_{50} profile machine. Each group has a computing potential of 2.5 volunteers of profile A_{100}. The results are depicted in Figure 5.8. As the results indicate, the *Charlotte* program is the only one able to maintain its efficiency: the efficiency of the *Charlotte* program degraded by approximately 5%; in contrast, the efficiency of RMI and PVM

programs dropped by as much as 60% and 45%, respectively.

5.6 Design Alternatives

There are two traditional approaches to implementing distributed shared memory at the software level. The most common approach (e.g., [90, 34, 22, 10, 2]) uses the hardware support for virtual memory, and requires assistance from the operating system. This is not a practical option for *Charlotte* because the JVM prohibits access to virtual pages. The second approach is to instrumenting the object code to detect and service shared memory read and write operations. The instrumentation can be done by a compiler at the source code-level [140], or by a rewriting tool at the binary-level [121]. If this approach was used for the first implementation of *Charlotte*, the extensions made to JVM [133] would have required changes to *Charlotte* before it could run again. Hence, instrumentation of the object code is also not a good solution.

To overcome the above limitations, *Charlotte*'s shared memory is implemented by shared objects. The flexibility of this technique, however, comes at a cost: accessing builtin Java data-types is orders of magnitude faster than accessing objects through method invocations. The experiments of Section 5.5 had very little data access and, hence, overheads due to method invocations are not visible. But, in general, this overhead can not be ignored.

Karl [79] has proposed a series of intermediate solutions to remove this overhead. He has shown that using a set of annotations to provide the runtime system with data access patterns, the entire read set can be propagated to worker pro-

cesses along with job assignments. This technique removes the method invocation overhead as well as the network latency due to page faults. Although this approach unnecessarily complicates the task a shared memory system attempts to simplify, it is a feasible alternative given JVM's lack of support for object management. However, as Java extend from a programming language to an operating environment, e.g., JavaOS [94], it is desirable to have the necessary object management support within the JVM. Such a support could enable a more efficient implementation of distributed shared objects.

Chapter 6

Middleware for Web-Based Applications

6.1 Introduction

The combination of the Java programming language and Web browsers' ability to load and execute untrusted Java applets in a secure sandbox has made distributed computing over the Web a possibility. Consistent with Java terminology, the term *applet* refers to a Java program that is typically loaded from the Internet and executes inside a browser. Browsers generally impose the *host-of-origin* policy [125] for secure execution of applets loaded from remote machines: browsers disallow applets from accepting and initiating network connections, except from and to the machine from which they were downloaded. In order to comply with this security model, distributed programs that utilize browsers and Java applets generally have the following characteristics:

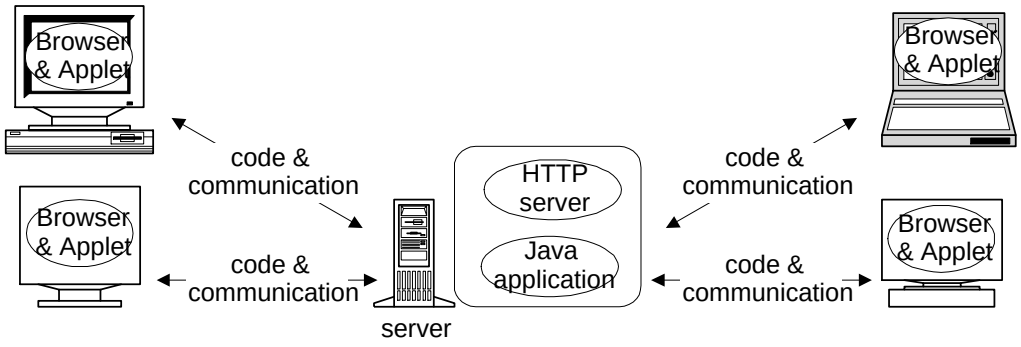


Figure 6.1: Architecture of a typical Web-based parallel or collaborative computation.

- The program is composed of a Java application and one or more Java applets.
- The Java application executes on the *end-user's* (defined in Section 5.1, page 102) local machine and is not under the scrutiny of the applet security model. The end-user's machine also runs an HTTP server to handle requests for applets.
- *Volunteers* (defined on page 102) are required to know *a priori* the URL address of the end-user's machine that is running the HTTP server.
- Once an applet loads and executes inside a volunteer's browser, it establishes network connections to its associated application. This application is responsible for transmitting information from one applet to another.

This is illustrated in Figure 6.1, and applies to parallel programming systems such as Javelin [31] and Bayanihan [120], and to collaborative applications such as Java Collaborator Toolset [83] and Jada/PageSpace [40].

In an ideal case, end-users should not be required to run an HTTP server on their machine; volunteers should not be required to have *a priori* knowledge of the

URL of the end-user's machine; and collaborative applets belonging to the same distributed application should be able to communicate directly. This is the goal of *KnittingFactory* middleware presented next.

Contributions

This chapter presents *KnittingFactory*, a Java middleware layer that facilities the development and execution of Web-based applications and extends the capabilities of high-level programming environments. The contributions of *KnittingFactory* are mechanisms for the following services:

- A distributed *directory service* to assist in finding Web-based applications on unknown hosts. This service is unique in utilizing the existing Web infrastructure to provide its service, and to migrate most of the computation away from the directory servers and onto Web browsers. It is designed to support applications that frequently register and deregister, such as parallel programs, and provides a non-uniform namespace that allows lookup operations to find an appropriately "close" computation.

- A light-weight *embedded class server* to eliminate the need for external HTTP servers and to provide a portal into applications and the means for applications to communicate with browsers via dynamic HTML files.

- A middleware service for direct *inter-applet communication*. This service is unique in making it possible for applets of the same distributed session, which are executing in multiple browsers on the Internet, to directly communicate and exchange information.

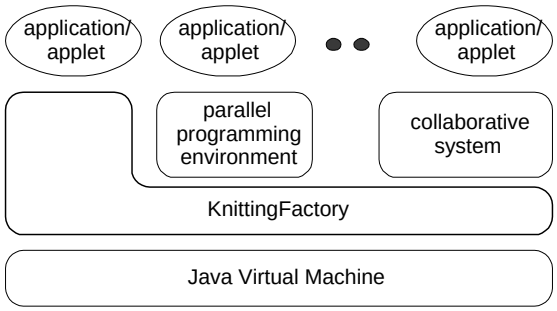


Figure 6.2: *KnittingFactory* in a layered design. It supports distributed applications and provides an infrastructure for other higher-level software systems.

Road Map

The rest of this chapter is organized as follows. Section 6.2 presents an overview of *KnittingFactory* and the architecture of the software system. The design and implementation of the three services, namely, the directory service, the embedded class server, and inter-applet communication, are presented in Sections 6.3, 6.4, and 6.5, respectively. Security concerns regarding inter-applet communication are discussed in Section 6.7.

6.2 Architecture

While many other systems provide high-level functionality to programmers and users, *KnittingFactory* is concerned with providing the infrastructure for such systems. *KnittingFactory* was initially designed to support high-level programming systems such as *Charlotte*, however, it can also be used by user applications. Figure 6.2 depicts its relation to user applications and programming environments. Readers interested in its application to collaborative programs and other user-

applications should consult [13]. *KnittingFactory* provides the following services:

- *KF Directory Service*: a distributed name service to assist users in finding networked applications on unknown hosts. This makes it possible for the user to find such applications on the Internet by performing a lookup at any site offering this service. The goal is to support applications which register and deregister frequently. The challenge is to provide this functionality in a decentralized fashion, but more importantly, in a way that works with "simple" clients such as Web browsers.

- *KF Class Server*: an embedded HTTP server to eliminate the need for external servers. A user who wishes to initiate a distributed application might not have access to a host running an HTTP server and it may be inconvenient or impossible to install such a server on the local host. This service provides a light-weight mechanism for users to run a distributed application on any host. In addition, this service provides a simple means and browsers to communicate through HTTP requests and dynamic HTML files.

- *KF Applet*: enables direct applet-to-applet communication. The host-of-origin policy imposed by browsers prevents direct network connections between applets. This service allows applets of the same distributed session, executing in multiple browsers on the Internet, to directly communicate and exchange information.

The design and implementation of these services are presented next.

6.3 Directory Service

Consider the scenario of dynamically changing set of managers (each in charge of a computation) and a dynamic set of available machines willing to run volunteers for managers. In this scenario, the difficulty lies in associating each volunteer machine with a computation. It is only when this association has been established, and the necessary program fragment has been made accessible to the volunteer machine, that the machine can contribute to a computation. The term *match-making* is used to refer to this problem. There are some obvious requirements for this match-making process: a simple API to work with Web browsers; a distributed architecture to achieve scalability, non-uniform namespace to keep parallel computations localized by assigning volunteers to nearby computations, and decentralized (i.e. having no single point of failure) to tolerate failures.

Motivation

The initial *Charlotte* implementation [15] did not address the match-making issue in a scalable and flexible fashion. At that time, *Charlotte* computations advertised with a single registry process, which would add an entry in a specific HTML file on their behalf. Volunteers directed their browsers to that HTML file for the list of active computations, then with a mouse click they could download and execute a cooperating applet. Figure 6.3 shows the Web site as volunteers would have seen it. This architecture was not scalable due to the single registry process, and it was also prone to process and network failures. In addition, volunteers required *a priori* knowledge of the registry's URL address. Following our initial work on *Charlotte*

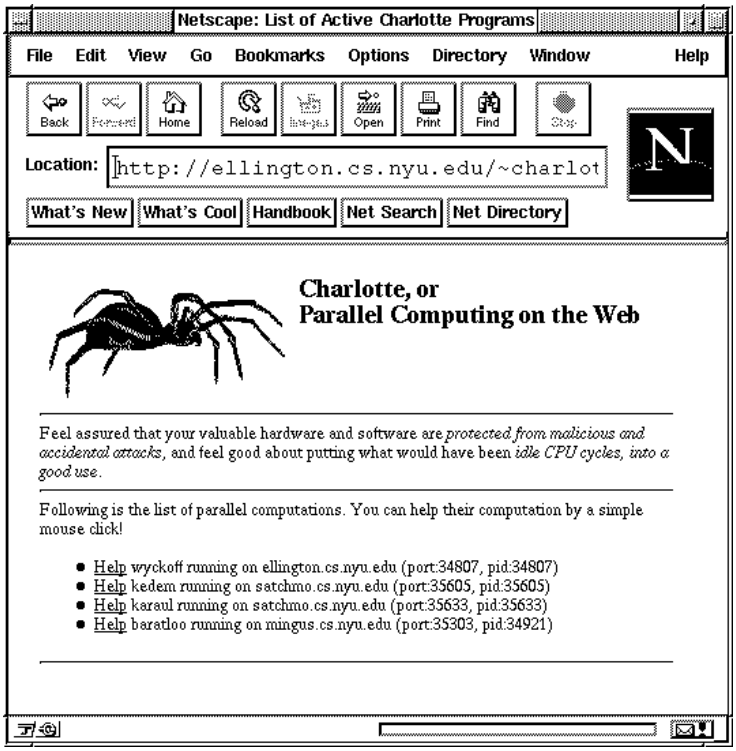


Figure 6.3: The registry service for the initial *Charlotte* implementation. Parallel computations are seen as entries in the HTML file; volunteers would chose a computation and click on the entry to participate.

there have been several other programming systems [31, 120, 30, 126] for Web-based volunteer computing. However in all of them, the solution for the match-making problem has the same set of limitations as our initial implementation.

In an ideal situation, the match-making should be performed by a distributed service: end-users that need computational help should register with a local process (either on the same machine or on the local network); volunteers should contact a local service that may be different from the ones end-users register with; and a directory service should assist in the match-making while respecting parallel computations' preference to execute as locally.

Design Goals and Considerations

While there are many classical directory service implementations, for example CORBA name servers [122], and many new proposals for a general purpose directory service, such as Lightweight Directory Access Protocol (LDAP) [134, 135], these are ill-suited in our context for three reasons.

- First, directory services such as LDAP define a uniform namespace: everyone has the same view of the data no matter where the request comes from. For example, if two volunteers, one in North America and one in Europe, search a uniform namespace for *Charlotte* computations, they would see the same set of computations. This is a desirable property for many applications but not for parallel computations. For efficient use of networks, the volunteer in North America should find and contribute to a programs executing nearby: on the same local area network if possible, otherwise in the same building,

the same city, and so on. Resources should not be wasted in providing a uniform namespace when it is not necessary.

- The second reason popular directory services are not well suited for our purpose relates to the nature of entries in the directory servers. Since parallel computations are dynamic in nature, a directory service must accommodate highly dynamic registration and deregistration of entries. The frequent registration and deregistration would make replication of information among distributed servers unattractive because it would also require frequent invalidations. But without replication, the server processes will have to perform a considerable amount of work to service lookup operations that don't succeed locally.

  For instance, LDAP directory servers are not allowed to redirect client requests [134], and hence, the servers are expected to chase down referrals. Thus, the CPU cycles and network bandwidth of the machine hosting the LDAP server is highly burdened. It would be beneficial to move some of the work from the server machines to clients' machine.

- Finally, existing directory services were not designed for Web-based computing from the outset, hence, they do not capitalize on the Web infrastructure. For example, system administrators are encouraged to install LDAP servers [129] where existing HTTP servers are able to provide the necessary services.

The lack of a general solution to address the deficiencies discussed above kept *Charlotte* (and other volunteer-based computing systems) from incorporating a directory service.

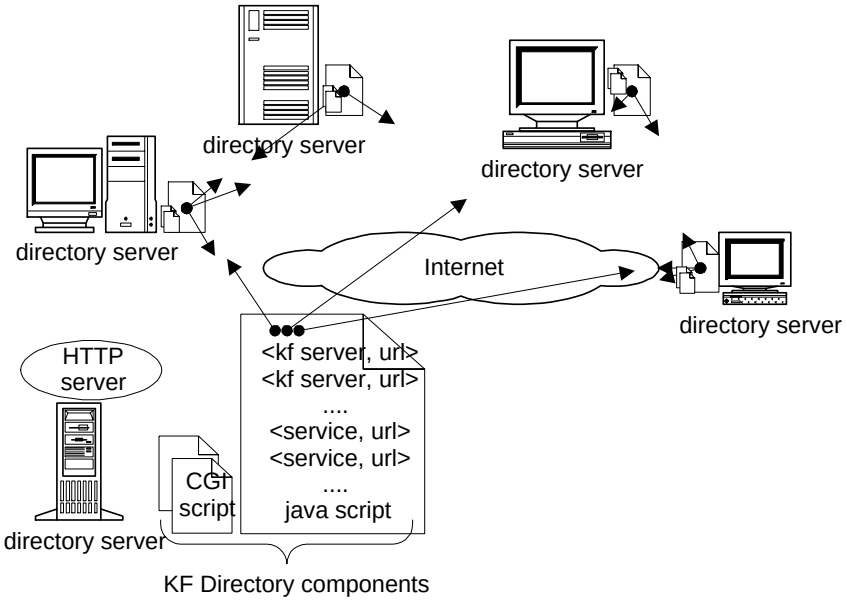


Figure 6.4: *KF Directory Service* components comprising a single HTML file and two CGI scripts.

Implementation

KF Directory Service capitalizes on the existing Web infrastructure. In particular, it does not introduce a process to act as a directory server; existing HTTP servers are used for this purpose. *KF Directory Service* is implemented by a single HTML file and two CGI scripts. To run an instance of *KF Directory Service*, a machine needs access to three files and it needs to run a standard HTTP server. Figure 6.4 illustrates the components of *KF Directory Service*.

The HTML file is logically separated into three regions. The first region contains a table of entries. Entries in the first region represent other directory servers. An entry is a tuple containing a string (i.e., service name); a URL address (location

of the service provider); and optional fields for category, password, and comments. For example, the directory server in the bottom of Figure 6.4 is aware of three other servers (represented by out-going edges), and hence the HTML file contains an entry for each of the servers. In this way the distributed servers form a directed graph. Although reachability (of every machine from every other) is a desirable property, there are no restrictions on the structure of the graph. Thus, users and administrators can customize the directory service to best suit their needs, for example, to reflect geographical proximity. The second region of the HTML file is also a table of entries as well, but the entries represent clients that have registered with this server (such as *Charlotte* programs). In practice, the first two regions are treated uniformly and stored in one table, and the category field is used to distinguish between entry types. The third region of the HTML file contains a JavaScript program, which performs the actual lookup. Note that while HTML files residing on different machines contain an identical JavaScript program, their directory entries generally differ.

The two CGI scripts are used to add and to remove entries in the HTML file. This design allows standard HTTP POST messages to serve as registration and deregistration requests. Furthermore, the uniform treatment of the entries makes it possible to dynamically add and remove table entries through Web browsers.

A lookup operation begins with a volunteer directing a browser to any one of the *KF Directory Service* sites and entering a request. This results in loading the HTML file containing directory entries as well as the JavaScript program. By being loaded, the JavaScript program starts executing and analyzes the entries on its own HTML file. If it finds an entry matching the search string, the browser

is instructed to load the corresponding link, which terminates the lookup. If the JavaScript program does not find a suitable match, the search must continue to one of the other sites listed, but the host-of-origin policy would prevent it from contacting other network sites. This problem is overcome as follows. If a match is not found, the JavaScript program constructs a new URL containing the address of the next server and its (program) state as a URL tag. The program state consists of a list of sites visited so far and a list of sites to visit next. The browser is then instructed to load the newly constructed URL. Loading the new HTML file causes execution of another identical JavaScript program and stops the execution of the current program, thus destroying its state. However, the newly executing program can reconstruct the previous program state by reading the tag field, and can continue to operate where the last program left off. This technique does not violate the limitations imposed by the *tainting* [56] (which prevents a JavaScript program from inspecting pages coming from other sites) and *host-of-origin* policies.

The lookup operation is interesting in its ability to move most of the computation away from directory-server processes and onto Web browsers: the actual search takes place in the user's browser, putting minimal strain on servers; the only service required from the server to send one HTML file. Furthermore, administration is reduced since the client-side code is downloaded automatically by the browser. Measurements (presented in Section 6.6) show that a successful lookup completes in approximately 500 milliseconds, and for unsuccessful lookups each hop accounts for about 200 milliseconds—this is the time offloaded from servers.

Application to Java RMI

This section presents how *KF Directory Service* can be used by systems other than volunteer based computing. In particular, *KF Directory Service* is applied to the client/server architecture of Java Remote Method Invocation (RMI) [52].

In Java RMI terminology, a *registry* is a remote interface for providing basic name server functionality. The `rmiregistry` program packaged in the JDK 1.1 is a shell script that invokes `RegistryImpl`, an implementation of the registry. Two methods provided by the registry are of special interest: `bind()` and `lookup()` for registering and locating a server object, respectively. The existing `RegistryImpl` binds a server object *only if it is local to its machine*. In the absence of a network-wide directory service, this means that client RMI applications must have *a priori* knowledge of the host running the server objects.

KnittingFactory can overcome this restriction by providing a directory service to RMI registries through a wrapper class around `RegistryImpl`. In addition to passing `bind` operations to the RMI registry, the wrapper class can add a corresponding entry in a *KF Directory Service* under a specified service name. A `lookup` operation can use the *KF Directory Service* to first search the network for the host running a registry with the appropriate server, and then to contact that registry for the server's remote reference. Thus, with the help of the *KF Directory Service*, RMI servers can execute anywhere on the network, and RMI clients can transparently find them without knowing which hosts to search.

6.4 Embedded Class Server

For volunteer computing, volunteer applets are loaded from an HTTP server and execute in a volunteer's Web browser. Volunteer applets typically establish a network connection to a process that coordinates their efforts. This is the case for applications written in *Charlotte*, Javelin, and the Java Collaborator Toolset among others. In the case of *Charlotte*, the coordinating process is the manager. The applet security model dictates that the manager process must run on the same machine as the HTTP server, which might not be convenient in some cases.

KnittingFactory addresses this problem by providing a light-weight HTTP server. *KF Class Server* implements the essential functionalities needed to serve applets, and is designed to be embedded into any Java application. As a result, it is the application itself that dynamically serves classes to browsers upon request. We had two goals in designing *KF Class Server*. First, it gives end-users the flexibility to initiate a *Charlotte* computation on any machine. Second, it serves as the means for applications and browsers to communicate through standard HTTP requests.

KF Class Server is implemented by the `KF_Server` class. A Java application can instantiate a `KF_Server` object to use its services. `KF_Server`'s constructor takes an applet's name (to include in the initial HTML file) and a call-back object (which will become clear below) as arguments. It then starts handling HTTP requests for `index.html` and Java class files; unrecognized requests are handed over to the application through the call-back object, and the response is sent back to the requesting browser. This gives applications a simple mechanism to communicate with external processes. In particular, *Charlotte* uses this feature

to provide a portal to the manager process, and allows browsers to monitor the progress of computations. Note that this is possible only because of the embedded nature of the HTTP server. The practicality of such a service, although simple, justifies its integration into *KnittingFactory*.

6.5 Inter-applet Communication

This section addresses facilitation of direct communication among multiple applets belonging to the same session and why this is beneficial. The security concerns regarding such inter-applet communication will be discussed in Section 6.7.

Motivation

The host-of-origin policy disallows applets (loaded over a network and) running on different machines from communicating through network connections. This hampers the "collaborativeness" of Web-based applications implemented by applets. Solutions to overcome this limitation rely on *untrusted code* or *a single forwarding agent*. Untrusted code, used either as native-methods [8] or browser plug-ins [17], is not subject to the host-of-origin policy and is free to establish network connections to any machine. The use of untrusted code, however, breaks the security guarantees provided by Java and prevents the use of off-the-shelf browsers. As an alternative method and as previously described on page 131, a Java application can be used as a forwarding agent to propagate information from one applet to another (see Figure 6.1). This technique has been used in several software systems [31, 83, 40], but as previously mentioned, the shortcomings of this approach

include a single point of failure and limited scalability.

In an attempt to alleviate some of the restrictions imposed by the host-of-origin policy, signed applets and certificates [125, 124] were introduced in Java 1.1. In addition, permission specifications and fine-grain access control will be added to a future release of Java [124]. Under the future model, the Java runtime system will maintain mappings from *code source* (i.e., the URL location and a corresponding certificate), to a *protection domain*, to user-defined permissions. Thus, users will be able to selectively override the host-of-origin policy by trusting certain code-providers. However, this requires code-providers to acquire a certificate, but contacting a certifying authority for a certificate is a tedious process for most casual application developers. *KnittingFactory* provides a flexible solution for inter-applet communication without burdening the user with the problematic choice of where to put trust, and without burdening the developer with the need to obtain a certificate.

Implementation

KnittingFactory's inter-applet communication is implemented through two Java classes: `KF_Registry` and `KF_Applet`. The `KF_Registry` class encapsulates an RMI registry service along with a *KF Class Server*. This way, browsers can request and get all the applets belonging to the same distributed application from a `KF_Registry` object. Consistent with Java terminology, a *remote object* is an object whose methods can be invoked from other Java virtual machines, and a *remote reference* is a reference to a remote object. `KF_Applet` is a remote object. Furthermore, `KF_Applet` extends the standard Java applet class as follows. First,

immediately after it is loaded in a browser and before the standard applet code has executed, it (1) registers its remote reference with the parent `KF_Registry`, and (2) gets a list of all other applets (siblings) that have registered with the parent `KF_Registry`. Second, through RMI calls it notifies its siblings of its intension to join the distributed application. After this bootstrapping phase, the `KF_Registry` is no longer needed—applets can use their mutually known remote references to communicate directly.

Sample Application

To demonstrate the viability of the *KF Applet* concept, a stand-alone whiteboard application was modified to work as a collaborative distributed application. The necessary changes were few and straightforward. First, the whiteboard applet was derived from `KF_Applet` instead of `java.awt.applet`. And second, AWT events were propagated to other members of the collaborative session using simple RMI calls to sibling objects.

6.6 Experiments

The following experiments were conducted on 200 MHz PentiumPro machines running the Linux version 2.0.34 operating system, and connected by a 100Mbps Ethernet through a non-switched hub.

An important performance characteristic of a directory service is the time to register and deregister servers, and the time observed by clients to perform a lookup operation. Six *KF Directory Services* (HTTP servers) were link together

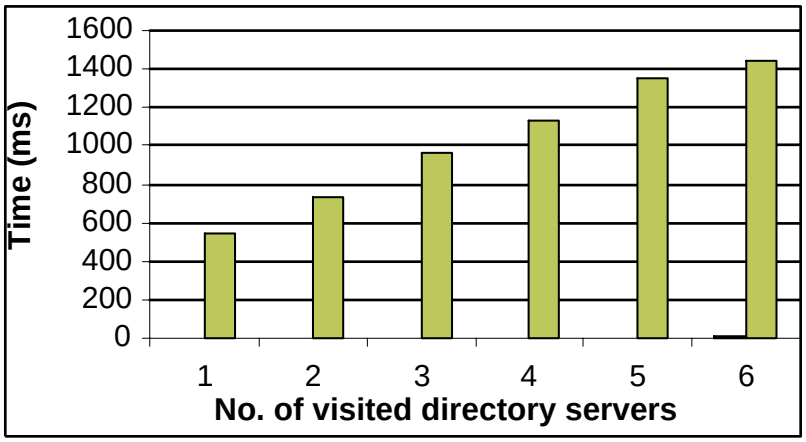


Figure 6.5: Lookup times for *KF Directory Service* in milliseconds. The x-axis shows the number of directories searched before the lookup operation succeeds.

in a configured in a chain-like structure. To measure registration and lookup times while abstracting away network delays, the *KF Directory Server* ran on the same machine as the browser that issued the commands. The measurements showed that registration of a new service takes 665 milliseconds and deregistration takes 685 milliseconds. The measured times for lookup operations are illustrated in Figure 6.5. The figure shows that a successful lookup completes in approximately 500 milliseconds, and for unsuccessful lookups, each hop accounts for about 200 milliseconds.

6.7 Security Concerns

Web browsers are expected to guarantee the execution of untrusted applet code in a safe manner. The host-of-origin policy, which is the most widely used security model, disallows applets from (1) initiating a network connection to any host other

than the one from which they came, and (2) listening for network connections initiated elsewhere. It is apparent that the RMI mechanisms used by *KF Applet* (tested using Java Developers Kit version 1.1.3, and HotJava Browser Version 1.1 running on Linux and Windows NT 4.0) violate the host-of-origin policy in providing direct applet-to-applet communication.

The following justification is given for the host-of-origin policy, "the intent is to prevent applets from using network connections to circumvent file protections or people's expectations of privacy" [123]. The host-of-origin policy also serves to protect machines located behind a firewall from an attack by an applet downloaded from the outside. As discussed earlier, the typical solution [23, 125, 74] to overcome the limitations host-of-origin is to use a stand-alone application to route messages between applets. *KF Applet* has been designed to work only for applets belonging to the same session. In some sense, this does not violate the spirit of Java, but only the specific technical mechanisms. This work demonstrates the usefulness of inter-applet communication, and it may be possible to formally and securely incorporate certain forms of inter-applet communication in future versions of Java.

Chapter 7

Related Work

7.1 Overview

Many research efforts have attempted to effectively harness computer resources in distributed platforms. Existing designs, however, separate certain related issues needed to achieve this goal. As a result, existing systems address issues such as programming ease, unpredictable machine behavior, fault-masking, and resource management, *in isolation*. This section provides a brief summary of ongoing research efforts and highlights in particular those research activities that are most relevant to the goals of this dissertation.

7.2 Parallel Computing on Networks of Workstations

Current distributed systems for networks of workstations fall into three general categories: message passing systems, remote procedure call systems, and distributed shared memory systems. This section discusses research related to each category

in turn, followed by several selected research projects.

Message Passing Systems

PVM [59], P4 [93], and MPI [67] are representative of message passing systems. Such systems typically provide a portable communication library and system independent mechanisms to start and control programs on remote machines. Furthermore, they provide synchronization primitives such as barriers and high-level communication services such as one-to-many and many-to-one messages. They are also very efficient.

Developing programs with message passing systems is made difficult mainly because of a lack of high-level support for program development. This is a common limitation with all message passing systems because they resemble the underlying hardware: they are low level, require the programmer to marshal and unmarshal data, and explicitly perform synchronization. Furthermore, the systems mentioned above do not provide any support for load balancing, fault masking, or comprehensive resource management, factors that limit their use as a metacomputing framework. For instance, PVM's failure detection is based on "best effort" without any guarantees. Nonetheless, PVM and MPI have become the most widely used parallel programming tools, primarily because of performance.

Remote Procedure Call Systems

Remote procedure call systems add structure to message passing. Concert/C [5], DCE-RPC and CORBA [19] are examples of mature and portable packages. In the context of a metacomputing framework, they suffer from all the shortcomings of

message passing systems. In addition, the synchronous nature of a procedure call conflicts with the notion of concurrency, further limiting this approach for parallel computation.

Distributed Shared Memory Systems

Another class of systems for distributed computing focuses on providing a virtual Distributed Shared Memory (DSM) across loosely-coupled machines. IVY [90], Midway [22], Munin [20, 34], TreadMarks [2], and Quarks [81] are representative of DSM systems. Traditional DSM systems suffer from performance problems, partially due to false sharing. In an attempt to improve performance, weaker and sometimes multiple memory-consistency semantics have been introduced. The decision as to which consistency model should be used, and when, is left to the programmer, thus complicating a task the researchers were seeking to simplify. Furthermore, none of the DSM systems mentioned above has architectural support for scalability, load balancing, or fault masking.

Linda [33] is a variant of DSM that provides a common global space. Piranha [60, 32] is a system built on top of Linda that allows machines to join an ongoing computation as they become idle and to retreat when reclaimed. Piranha, however, can not tolerate failures. With the exception of limited load balancing, Linda and Piranha suffer from all the shortcomings of DSM systems, as well as some of the shortcomings of message passing systems since programmers have to marshal and unmarshal data in tuples. Linda and other related systems are further described below.

Fault Tolerant Systems

Fault-tolerant systems such as Isis/Horus [25, 132], Ensemble [131], and Transis [48] utilize reliable and ordered multicast to provide virtual synchrony: an environment to mask unpredictable asynchronous network behavior. Programming, however, is based on message passing, and the programmer is responsible for dealing with failures once they are detected. In such systems, the overhead for fault tolerance is high, and this overhead is present even in the absence of failures.

Other work aimed at providing fault tolerance has mainly involved augmenting existing systems with conventional techniques, such as check-pointing and process migration (FS-PVM [88]), transaction wrappers (PLinda [77]), and active replication (FT-Linda [7]). In such systems, tolerating failures is never transparent and requires a special programming effort.

Combined Approaches

Several projects address related issues without specifically targeting the goals of our research. The Berkeley NOW project [108] aims to utilize all resources on a network. They target specific areas: using of low-latency active messages for fine-grain data sharing; using the aggregate DRAM on a network as a giant disk; using software implementations of RAID to increase data bandwidth and to remove single points of failure; and utilizing idle CPU cycles. The Wisconsin Wind Tunnel [113] provides a middle-level communication interface. Tempest [71] is a low-level communication substrate for implementing message passing, shared memory, and hybrids of the two.

Linda and Friends

Linda programs communicate using *tuples*. A tuple is a sequence of typed values. For example, `("hello", "world")`, and `(10.0, 42)` are tuples. A *template* is a sequence consisting of types and typed values. For example, `("hello", ?cPtr)`, and `(?f, 42)` are templates. The fields of a template whose values are specified are *actuals* and the other fields are *formals*.

A *tuple space* is a virtual shared bag that houses tuples. Programs communicate by placing and removing tuples from the tuple space. Six operations are provided for this purpose. A Linda process can add a tuple to the tuple space using the `out` operation, remove a tuple form the tuple space using the `in` operation, and query the tuple space to determine the existence of a tuple using the `rd` operation. The `in` and `rd` return a tuple, if one exist, from the tuple space that matches the type of the template. Both `in` and `rd` operators are synchronous and block until a matching tuple appears. There might be times when the synchronous behavior is not desirable. The `inp` and `rdp` are asynchronous operators analogous to `in` and `rdp`, respectively. There is a special operation for process creation. The `eval` operator takes a template as argument and creates a processes for each actual argument. When all the spawned processes have terminated, the tuple is created and put in the tuple space.

Linda

Linda is highly tuned for performance. It comprises of two separate software modules: a preprocessor and a runtime library.

The Linda preprocessor replaces the six operations (discussed above) with appropriate function calls to the Linda runtime library. A major portion of performance optimizations occur at this stage. The preprocessor statically analyzes the program to spatially partition the tuples and the templates into *equivalent class*. This way, each operation can be targeted to a specific tuple space partition. The preprocessor also determines the best data structure to use for each equivalent class. For example, if a partition consists of only `(''work'')` tuples, then the tuples will be represented as a single integer counter.

The Linda runtime library implements the six operations. The initial implementation was based on a centralized tuple server; distributed implementations were introduced later. An important performance question that Linda addresses well is where to store the tuple space. The following choices are viable: keep tuples local to the machine that generates them, replicate tuples across all machines, or have a home machine for each equivalent class. Linda tries to utilize all three approaches, however, the latter is the most widely used.

Piranha

Piranha enables Linda programs to run on idle workstations. It allows workstations to join an ongoing computation as they become idle and to retreat when reclaimed by their users. The Piranha encourages programmers to write their program in a restricted form of the master/worker paradigm, whereby each Piranha process reads a work tuple, does some computation, outputs a tuple, then dies. Unlike Calypso, Piranha does not address fault tolerance and other unpredictable machine characteristics that arise from time-sharing.

Persistent Linda

Persistent Linda (PLinda) adds fault-tolerance to Linda by using atomic transactions. It extends the standard six Linda operations with `xstart` and `xcommit` to indicate the start and end points of a transaction, and `xrecover` for process recovery. The `xcommit` operation stores the local state of a process (continuation) in the tuple space. This continuation tuple can be read by a recovering process to restore its state. Database transactions are generally known to be heavy weight and not appropriate for parallel computations. To address this problem, PLinda uses lightweight transactions (level-two commit protocol). Furthermore, the `xcommit` operation is an in-memory operation—the idempotence nature of parallel computations makes this an attractive solution. Fault tolerance in Calypso, unlike PLinda, is transparent to programmers and the executing program.

Dome

Distributed Object Migration Environment (Dome) [3, 18] is notable because ease of use, load balancing, and fault tolerance were address at the onset. Dome provides data-level parallelism through a set of pre-defined C++ classes libraries and runs on top of PVM.

Load balancing is achieved as follows. When a dome object is instantiated, its data is partitioned and distributed among participating machines. The system then tracks the speed of each machine and dynamically repartitions the data to achieve a balanced load. Fault tolerance is addressed through checkpointing. In what is called the high-level heterogeneous checkpointing, programmers are responsible for

explicitly storing program states and later for recovering to a consistent state. This is the only implemented technique. The low level checkpointing consists of regular core dumps. In this case, determining a consistent state is a difficult issue since the failure could have occurred during process communication. Dome's architecture allows it to be portable and heterogeneous. In fact, there are several different ports Dome.

Programming effort is kept to a minimum as long as the Dome's pre-defined objects suffice. Otherwise, for user-defined objects to enjoy the benefits of load-balancing and fault-tolerance, they must abide by and mesh into the rest of the Dome runtime system. This means that the programmer must use PVM communication mechanisms to partition, distribute, and redistribute data, and to implement checkpointing. This extra effort requires specialized programming, which limits Dome's use. The effectiveness of Dome's load-balancing technique is limited as well. According to published results, an experiment was conducted to evaluate load balancing [3, 18]. When one slow machine (slowed down by a factor of 67%) was added to five other machines performing a large matrix multiplication operation, *performance dropped* by almost 50%.

Cilk-NOW

Cilk-NOW [28] is a software system designed to run parallel programs on networks of workstations. A macroscheduler, which schedules Cilk processes on idle workstations, provides Cilk-NOW programs with adaptive parallelism. A centralized resource manager called the job broker implements the macroscheduler.

The Cilk-NOW runtime system supports a subset of the parallel Cilk [26]

language—a multithreaded extension of C. A Cilk program contains one or more Cilk procedures, and each procedure contains one or more Cilk threads. A Cilk procedure is the parallel equivalent of a C function, and a Cilk thread is a non-suspending piece of a procedure. A Cilk program (procedure) achieves parallelism by spawning successor and child threads: typically, child threads are used for parallelism, and successor threads are used by parent threads to synchronize with child threads. Threads communicate through arguments and return values only: Cilk-NOW does not provide distributed shared memory on networks of workstations.

Cilk, and similarly, Cilk-NOW, uses the "work stealing" algorithm [27] to schedule threads on participating machines. The work-stealing scheduler works as follows: a process with no (runnable) threads randomly steals threads from neighboring victim processes. The work-stealing algorithm is provably efficient and predictable for identical speed processors.

Cilk-NOW provides transparent fault tolerance as follows. First, Cilk-NOW ensures that subcomputations execute in isolation, i.e., the work done by one subcomputation is not visible by others until the subcomputation finishes and its result is returned. Second, once a failure is detected, other remaining processes check to make sure that they have not been an earlier victim of the failed process. If a process detects that it has been a victim, it assumes responsibility for executing the computations that were stolen (and never finished) due to the failure. Additionally, each surviving process checks the list of computations assigned to it, and removes any computations that were stolen from the failed process. This step also involves informing other processes that might have stolen subcomputations

of aborted computations. Finally, the computations lost due to the failure are recovered from the last checkpoint image. Checkpointing occurs periodically and is transparent to Cilk programs.

Cilk-NOW provides adaptive parallelism, automatic load balancing, and fault tolerance. In contrast to Calypso, it does not provide a virtual shared memory, and its scheduler was not designed for processors with different and unpredictable speeds.

7.3 Parallel Computing on the World Wide Web

The Use of the Internet to form large metacomputers out of geographically distributed machines has resulted in a number of research projects with various points of focus.

The Legion system [65] attempts to create a single virtual computer out of distributed computers, leveraging diverse technologies such as object-oriented programming, wide-area gigabit networks, cryptography and parallel compilers. Globus [57] targets computational grids for providing dependable, consistent, and pervasive access to high-end computational resources. Legion provides a highly organized view of the distributed system, that is, every component is represented by a corresponding object. Globus, on the other hand, follows a bag of services architecture, and provides services such as resource management and security, from which developers can select based on their needs'. Legion and Globus have a much larger focus than *Charlotte*. As a result, they do not address issues like ease of programming, load-balancing and adaptive execution of programs, and

distributed shared memory, all of which are major concerns in *Charlotte*.

Important concerns for distributed computing systems include security, system heterogeneity, and program distribution. A considerable number of systems concentrate on using Java to solve some or all or of these problems. These systems can be broadly classified into two categories: systems that use off-the-shelf Java compilers and Java Virtual Machines (JVM); and systems that modify and extend JVMs. An orthogonal classification can also be made according to the programming model that is offered. In particular, message passing or DSM systems are of interest due to their close relationship to programming models found for parallel computing. Metacomputing is in principle possible using distributed object technologies, such as those offered by Voyager [105], Aglets [86], and Java Remote Method Invocation (RMI) [52]. However, these systems do not target metacomputing; rather, they provide a starting point to build higher-level systems for metacomputing. These systems are not therefore directly comparable to a metacomputing project such as *Charlotte*.

Examples of projects that do not rely on Pure Java are ATLAS [8], IceT [64], Java/DSM [139], and ParaWeb [29]. ATLAS adapts Cilk's work stealing mechanism to the Web. It implements a tree-like hierarchy of managers on top of (volunteer) compute servers. In addition to having compute servers steal work from one another, the managers move work from one subtree to another for inter-cluster load balancing. This architecture is similar to *KF Directory Service* in that the computations tend to stay localized. However, unlike *KF Directory Service*, the managers perform the lookup searches, and the Web-infrastructure is not utilized. More importantly, because the managers are an integral part of ATLAS's programming

model, it is not clear how this architecture can be applied to other programming systems. Furthermore, ATLAS uses native code to circumvent the Java sandbox security model. IceT is concerned with integrating legacy and native-language libraries into Java programs while still offering code mobility. Java is used as a wrapper language for portability and existing native-language libraries are used for the actual computational engine of an application. Security concerns are addressed via a special analysis of the Java bytecode. Java/DSM modifies the JVM to interface with the Treadmarks DSM system. All Java objects are allocated in a shared memory region, and the garbage collector is modified to maintain conservative estimates of cross-machine object references. The use of the page-based Treadmarks system complicates maintaining heterogeneity and requires modification of the Java object-layout. ParaWeb modifies the JVM to allow Java threads to be instantiated on any remote machine and adds a release-consistency DSM abstraction to implement a single global namespace across threads. The basic Java synchronization mechanisms are used as synchronization points for the consistency mechanism.

Previously mentioned systems rely on native-code or non-standard JVM implementations to provide a parallel programming environment. While this approach allows the fine-tuning of communication and memory management systems, it does not maintain Java's secure execution environment and it requires installation of additional software. Therefore, such an approach may not be applicable or desirable in many circumstances; specifically, because it breaks the basic requirements of one-click computing.

JPVM [55] and mpiJava [6] provide a message passing interface on top of s-

tandard Java. These two software systems provide the same benefit to programs written in Java that PVM and MPI (respectively) provide to programs written in C: they provide a set of low-level mechanisms to send and receive messages. Unlike *Charlotte*, those software systems do not provide any support for load-balancing and fault-masking, and more importantly, programs written in JPVM and mpiJava can not execute as Java applets, which makes them ill-suited for Web-based computing.

JavaParty [109] and Ninflet [126] are Java-based systems for distributed computing. JavaParty provides mechanisms (built on top of Java RMI) for transparently distributing remote objects. Ninflet is an infrastructure for migrating objects, which targets parallel computing on idle CPU cycle. Unlike *Charlotte*, JavaParty and Ninflet programs are Java applications, and hence they are not able to execute in Web browsers. Furthermore, they do not transparently handle faults and they do not load balance parallel applications.

Javelin [31] and Bayanihan [120] are systems that address volunteer-based computing. In Javelin a standalone application, called the broker, acts as the central task repository and scheduler. Javelin provides multiple communication models, including PVM and Linda models, and the broker is used to forward messages from one volunteer to another. In contrast, *Charlotte* provides a predictable virtual machine model and a runtime system that implements this machine on the unpredictable machines of the Web. Bayanihan is similar to *Charlotte* in the services it provides for volunteer-based computing. The two major components of the Bayanihan system are the communication module and the scheduler. Currently, Bayanihan provides communication in the form of migrating objects, which are

implemented on top of the HORB [72] system. The current implementation of Bayanihan uses eager scheduling, and thus is able to provide transparent fault tolerance and load balancing like *Charlotte*.

Since the initial work on *Charlotte*, there have been several other programming systems [31, 120, 30] that strive to realize one-click computing. In every case, their solution for matching volunteers with computations, i.e., match-making, has the same limitations as the initial implementation. For example, in the case of Ninflet, *server daemons* (processes that volunteer to do work) and *client processes* (computations that need help) are given the URL address of a *dispatcher* by initiating users. The dispatcher is a process responsible for assigning computations to servers. Thus, Ninflet relies on a single process for match-making, and assumes the user's *a priori* knowledge of the process' URL address. This is also true for Javelin and POPCORN [30] (where the centralized match-making process is called the *broker* and *market*, respectively). Distributed match-making is not addressed in Bayanihan. *Charlotte*, through the use of *KF Directory Service*, is the first Web-based metacomputing system to overcome this limitation.

7.4 Overview of Selected Resource Management Systems

Load Dispersion for Sequential Jobs

Early systems for resource management were designed to disperse the execution of jobs among available machines on a network. To use their services, users submit programs to the underlying system for execution. The system then selects the machines to execute the programs. Executing programs' input and output are

generally redirected to provide the user with the output of the remotely executing program. Systems like NEST [1], V [127], Sprite [50], MOSIX [9], and Remote Unix [91] provide such a service at the operating system level. Coshell [58] provides a similar service, not as a part of the operating system but as a Unix shell. The only privilege Coshell requires is `rsh` access to remote hosts.

To manage the transient availability of machines, systems such as Remote Unix, Sprite, and MOSIX utilize checkpointing and process migration to move processes once machines become unavailable. Coshell provides user-level process migration through CosMiC [39].

In contrast to ResourceBroker, the focus of the above systems is to support sequential computations, and they do not make any special provision for parallel programs.

Static Allocation of Parallel Jobs

Several research and commercial products such as Condor [92], Utopia [141] (now LSF [110]), DQS [53] (now CODINE [61]), Portable Batch System (PBS) [68], and IBM's LoadLever [76] were developed for managing heterogenous resources of networks of workstations. These systems are typically Queue Management Systems and were originally intended to be used with batch sequential jobs. With the increased popularity of parallel programming systems such as PVM and MPI, previously mentioned systems extended their support to parallel interactive jobs. Globus [57] and Legion [38] are large-scale resource managers designed to unite machines from multiple administrative domains. While well-suited to what they do, the above systems are in some sense batch-like and limited to a static allocation

of resources.

To use the above systems, a user submits a job to the resource manager and specifies the number and type of machines required. The resource manager then selects a set of appropriate machines, starts up daemon processes, and performs other necessary tasks to prepare the parallel job for execution. In some cases the resource manager even starts and monitors the job, and then informs the user when the job completes. An important characteristic of such resource managers is the inherently static allocation of machines to jobs: machines are allocated only once and before the jobs starts executing. This is a limiting factor. In a recent study at the NASA Ames Research Center [78], six commercial and research resource managers were evaluated. The researchers in this study reported "[...] the bad news is the confirmation of a continuing lack of JMS [Job Management System] support for parallel applications, parallel systems, and clusters of workstations."

Attempts have been made to dynamically load-balance parallel jobs using techniques employed for sequential jobs: process checkpointing and migration. Resource managers such as GLUnix [62], PRM [103], and DynamicPVM [45] were designed to use this technique. Under such systems, when a machine becomes busy, processes executing on that machine are migrated to an underloaded remote machine. Although some researchers have argued that process checkpointing and migration are not effective for parallel jobs [4, 87], the literature is still undecided on the subject [51].

A dynamic allocation of machines to jobs is more effective if machine-loads, as well as job resource requirements, can change over time. In contrast to previous systems, ResourceBroker can dynamically allocate and reallocate resources to jobs.

Systems such as Piranha [60, 32], Cilk-NOW [28], MPVM [35], Condor/CARMI [111], and DRMS [100] specifically target adaptive parallel computations, and are capable of dynamically allocating machines to a running job as resources become available as well as deallocating machines if they are needed elsewhere.

Piranha and Cilk-NOW are examples of a resource manager integrated with a parallel programming system. Piranha is an extension of Linda, and Cilk-NOW is an implementation of Cilk. Both allow jobs to expand their execution to remote idle machines, and to retreat when machines are reclaimed. Piranha represents a great step forward in providing adaptive parallelism. However, it required modifications to the Linda system, and only supported Linda programs that had been modified to use Piranha. Similarly, Job Broker, the Cilk-NOW's resource manager, can only support adaptive Cilk programs.

Condor/CARMI proposed that "[...] all RM [resource management] functionality be removed from PPE [parallel programming environments] code, and migrated into one or more processes which are dedicated to handling resource management requests." This proposal lead to major modifications of version 3.3 of PVM's intra-job management services: the functionalities were isolated and removed form PVM daemons to made into PVM tasks. As a result, PVM now supports a well-defined interface for intra-job management. Both Condor/CARMI and MPVM utilized this interface to support adaptive PVM programs. However, the approach taken by Condor/CARMI and MPVM is PVM-specific, required

assistance from the implementers of PVM, and can not be applied to other programming systems such as MPI.

Distributed Resource Management System (DRMS) is a service that runs on the IBM SP2 computer and manages MPI and HPF jobs (that are compiled to use MPI at a lower level). It is parallel application aware, in that it can grow and shrink a job's "node count." However, it is tightly integrated with the SP2 system and the MPI implementation (it modifies routing tables to redirect MPI messages). Its dynamic services are also limited to programs that are explicitly programmed for DRMS.

The above mentioned resource managers are tightly integrated with either the underlying operating system or with the parallel programming system they support. In fact, unlike ResourceBroker, none of the systems support adaptive programs written using different parallel programming system.

Chapter 8

Conclusions

8.1 Metacomputing on Networks of Workstations

External factors that exist in multiuser networks of workstations give rise to unpredictable machine and network behavior. To efficiently use such environments, programs must adapt to these changes. A common weakness in most parallel programming tools is the lack of high-level support for developing adaptive programs, and hence, programmers are left with responsibility for this task. Chapters 3 and 4 presented Calypso and ResourceBroker which in unison provide a comprehensive solution for metacomputing on networks of workstations. Several novel ideas have been validated by Calypso, including *Eager scheduling*, *two-phase idempotent execution strategy* (TIES), *bunching*, aggressive shared-memory caching techniques, and adaptive scheduling policies. ResourceBroker has validated a technique for managing adaptive computations that is based on intercepting and interpreting low-level actions performed by computations.

Eager scheduling, *(TIES)*, and *bunching* were shown to be effective in providing transparent load balancing and fault tolerance. The interplay of eager scheduling and TIES result in distributing the work load on the bases of processor speeds. In addition, because faster machines are able to bypass slower machines, the addition of a slow machines to a parallel computation will never be detrimental to the performance—this is not the case with most other programming systems. Furthermore, the combination of eager scheduling and TIES allows a failure to be viewed as a special case of an infinitely slow machine. *Bunching* demonstrated how the eager scheduling and TIES can be extended to efficiently execute fine-grain tasks in a coarse-grain fashion.

In addition to the above techniques, it was shown that the combination of aggressive shared-memory caching techniques with adaptive scheduling policies can be used to efficiently implement a virtual machine on networks of workstations. Calypso's *caching* and *prefetching* algorithms leverage the structure of fork/join programming style to minimize the number of (false) page-faults and to amortize the overhead associated with fetching shared data over the network. The caching algorithm along with CREW programming model alleviates page-shuttling. Furthermore, Calypso's scheduler assists the caching mechanisms by attempting to schedule jobs based on its estimate of spatial and temporal locality of shared data.

The contributions resulting from Calypso being an operational system for several years are satisfying. A number of applications have been parallelized using Calypso, including from the general areas of graphics, image processing, finance [11], scientific [11], and computational physics. Calypso has also been used as a teaching tool in a graduate seminar course at New York University. Furthermore, other re-

searchers have used Calypso as a stepping-stone for other original work. Examples include:

- Huang *et al.* [73] showed that nested parallelism, i.e., a parallel step inside another parallel step, allows programming flexibility and implemented a system to demonstrate this. That work had its roots in Calypso.

- The experiences and lessons learned during a successful port of Calypso to Windows NT were reported in [43, 96].

- Sardesai *et al.* [118, 119] used the core mechanisms of Calypso (i.e., eager scheduling and TIES) in developing Chime. Chime is an elegant programming system, which among others, supports nested parallelism, distributed cactus stacks, inter-routine communication, and it implements the CC++[36] programming language on networks of workstations.

- In the context of Chime, McLaughlin *et al.* [95, 97] extended the bunching mechanism of Calypso with thread migration and preemptive scheduling for improved performance and resilience to failures.

The effective utilization of transient resources relies on resource manager's ability to communicate resource availability to adaptive computations, and adaptive computations' ability to communicate their resource requirements to the resource manager. Unfortunately there is no standard interface for this two-way communication. Hence, existing resource managers did not support adaptive programs written with different parallel programming system. Chapter 4 presented a technique to effectively built this communication interface, even to programs that were

not developed to work with resource managers. The technique is based on intercepting and interpreting low-level actions common to many parallel programming systems. Indeed, our initial goal was to support PVM and Calypso programs; only after the initial design we realized that MPI and PLinda programs can be supported as well.

Chapter 4 also presented the design and implementation of ResourceBroker, a resource manager to demonstrate the above technique. ResourceBroker is the first resource manager that can dynamically allocate, deallocate, and reallocate machines to adaptive programs written using PVM, MPI, Calypso, and PLinda. The use of external plug-in modules enables future support for as yet undefined programming systems. Furthermore, ResourceBroker executes with user-level privileges, and hence, it does not compromise the security of networked machines even if it malfunctions. The focus of this work, so far, has been on a general mechanism to coerce unmodified programs in using suitable machines at runtime. This is a starting point for more extensive research on allocating general resources to unmodified programs. The flexibility is needed in order to provide Quality-of-Service guarantees that enhance the execution of users' programs.

8.2 Metacomputing on the World Wide Web

Chapters 5 and 6, respectively, presented *Charlotte* and *KnittingFactory*, which in unison, provide a comprehensive solution for metacomputing on the Web. *Charlotte* leverages Web browsers' abilities to load and execute untrusted Java applets in a secure fashion to assist programmers with developing parallel programs and

a runtime system for the execution of the programs on the Internet. *KnittingFactory* is an infrastructure that facilities the execution of collaborative and parallel applications. The lessons learned by having an operational system are summarized below.

Users wanting to execute programs on the Internet can not be expected to have access-right to every machine. Therefore, *Charlotte* was based on the concept of (1) *volunteers* who would allow their machines to participate in someone else's computation, and (2) *one-click computing*, which allows volunteers to participate in ongoing computations by directing a Java-capable browser to a Web site. *Charlotte* is the first such system.

The success of projects that use the Internet for executing compute-intensive programs (described in Section 5.1) indicates that a large number of people will voluntarily donate the use of their machines. Thus, the concept of volunteer-computing seems to have been a good choice in designing *Charlotte*.

Several other systems also implement one-click computing, but to the best of my knowledge, and for valid reasons, one-click computing has not been used in solving any "real world" problems. One-click computing is implemented in Java and relies on the browser's ability to download and execute Java applets. Thus, the success of one-click computing relies on the acceptability of Java as a language for compute-intensive programs. It is generally agreed that current implementations of Java produce unacceptably slow programs [42]. It is possible that with continued improvements of Java compilers and virtual machine implementations [63, 66], Java and, in turn, one-click computing will become a viable choice for compute-intensive programs.

The initial implementation of *Charlotte*, as well as subsequent volunteer-based computing systems [31, 120, 30, 126], neglected the problem of matching volunteers with computations. *KnittingFactory* addresses this problem by providing a directory service. This directory service supports a non-uniform namespace to keep parallel computations localized, and it implements a novel technique to migrate most of the computations to the client side. In addition to the directory service, *KnittingFactory* provides the means to implement portals into *Charlotte* managers. As a result, standard browsers can be used to monitor the progress computations. *KnittingFactory* has contributed to the usability of *Charlotte*, and thus far, it been successful in meeting its goals. But it is not tied to *Charlotte* in any way; it will be gratifying to see *KnittingFactory* applied to other systems as well.

Mechanisms such as *eager scheduling*, *two-phase idempotent execution strategy*, and *bunching* are used for load balancing, fault masking, and efficient execution of fine-grain tasks. These mechanisms were first implemented in Calypso and proved effective for networks of workstations. Experiments with *Charlotte* indicate that these mechanisms are also effective for the Web, an environment that is more prone to failures and more unpredictable than networks of workstations. These mechanisms have since influenced other research projects, such as Bayanihan [120]—which is gratifying.

Bibliography

[1] R. Agrawal and E. Ezzat. Location independent remote execution in NEST. *IEEE Transactions on Software Engineering*, 1987.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 1996.

[3] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[4] R. Arpaci, A. Dusseau, L. Vahdat, L. Liu, T. Anderson, and D. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *SIGMETRICS*, 1995.

[5] J. Auerbach, A. Goldberg, A. Gopal, M. Kennedy, and J. Russell. Concert/C: A language for distributed programming. In *Proceedings of the Winter USENIX Conference*, 1994.

[6] M. Baker, B. Carpenter, G. Fox, S. Ko, and X. Li. mpiJava: A Java interface to MPI. In *Proceedings of the ACM Workshop on Java for High Performance Network Computing*, 1998.

[7] D. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[8] J. Baldeschwieler, R. Blumofe, and E. Brewer. ATLAS: An infrastructure for global computing. In *Proceedings of the ACM SIGOPS European Workshop*, 1996.

[9] A. Barak, S. Guday, and R. Wheller. *The MOSIX distributed operating system—load balancing for Unix.* Lecture Notes in Computer Science, Springer-Verlag, 1993.

[10] A. Baratloo, P. Dasgupta, and Z. M. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceedings of International Symposium on High-Performance Distributed Computing (HPDC)*, 1995.

[11] A. Baratloo, P. Dasgupta, Z. M. Kedem, and D. Krakovsky. Calypso goes to wall street: A case study. In *Proceedings of the International Conference on Artificial Intelligence Applications on Wall Street*, 1995.

[12] A. Baratloo, A. Itzkovitz, Z. M. Kedem, and Y. Zhao. Mechanisms for just-in-time allocation of resources to adaptive parallel programs. In *Proceedings*

of the International Parallel Processing Symposium & Symposium on Parallel and Distributed Processing (IPPS/SPDP), 1999.

[13] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. An infrastructure for network computing with Java applets. In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*, 1998.

[14] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. An infrastructure for network computing with Java applets. *Concurrency: Practice and Experience*, 1998.

[15] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1996.

[16] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *To appear in International Journal on Future Generation Computer Systems*, 1999.

[17] L. Beca, G. Cheng, G. Fox, T. Jurga, K. Olszewski, M. Podgorny, P. Sokolwski, and K. Walczak. Web technologies for collaborative visualization and simulation. Technical Report SCCS-786, Northeast Parallel Architectures Center, 1997.

[18] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing on Workstation Clusters and Network-based Computing*, 1997.

[19] R. Ben-Natan. *Corba.* McGraw-Hill, 1995.

[20] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1990.

[21] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). RFC 1738, 1994.

[22] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proceedings of the International Computer Conference (COMPCON)*, 1993.

[23] D. Bhatia, V. Camuseva, M. Camuseva, G. Fox, W. Furmanski, and G. Premchandran. Web-Flow—a visual programming paradigm for Web/Java based on coarse grain distributed computing. *Concurrency: Practice and Experience*, 1997.

[24] N. Biggs. *Interaction models: Course given at Royal Hollaway College, University of London.* Cambridge University Press, 1977.

[25] K. Birman. Replication and fault-tolerance in the ISIS system. *ACM Operating Systems Review*, 1985.

[26] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *The Journal of Parallel and Distributed Computing*, 1996.

[27] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, 1994.

[28] R. Blumofe and P. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX Annual Technical Conference*, 1997.

[29] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards world-wide supercomputing. In *Proceedings of the ACM SIGOPS European Workshop*, 1996.

[30] N. Camiel, S. London, N. Nisan, and O. Regev. The POPCORN project—an interim report: Distributed computation over the internet in Java. In *Proceedings of the International World Wide Web Conference*, 1997.

[31] P. Cappello, B. Christiansen, M. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 1997.

[32] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and Piranha. *Computer*, 1995.

[33] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 1989.

[34] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, 1991.

[35] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walope. MPVM: A migration transparent version of PVM. *Computing Systems*, 1995.

[36] M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 1992.

[37] M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. *Lecture Notes in Computer Science*, 1993.

[38] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource management in Legion. *International Journal on Future Generation Computer Systems (to appear)*, 1999.

[39] E. Chung, Y. Huang, and S. Yajnik. Checkpointing in CosMic: A user-level process migration environment. In *Proceedings of Pacific Rim Symposium on Fault-Tolerant Computing*, 1997.

[40] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Redesigning the Web: From passive pages to coordinated agents in PageSpaces. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, 1997.

[41] R. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *Computer Graphics*, 1984.

[42] M. Curtin and J. Dolske. A brute force search of DES keyspace. *;login:*, May 1998.

[43] P. Dasgupta. Parallel processing with Windows NT networks. In *The USENIX Windows NT Workshop*, 1997.

[44] E. Dijkstra. Solution of a problem in concurrent programming. *Communications of the ACM*, 1965.

[45] L. Dikken, F. van Der Linden, J. Vesseur, and P. Sloot. DynamicPVM: Dynamic load balancing on parallel systems. In *Proceedings High-Performance Computing and Networking*, 1994.

[46] distributed.net. *Project Bovine.* Available at `http://www.distributed.net/rc5`.

[47] distributed.net. *Project Monarch.* Available at `http://www.distributed.net/des`.

[48] Dolev and Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 1996.

[49] H. Donald and M. Baker. *Computer Graphics, C version.* Prentice Hall, 2nd edition, 1997.

[50] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience*, 1991.

[51] A. Downey and M. Harchol-Balter. A note on "the limited performance benefits of migrating active processes for load sharing". Technical report, University of California at Berkeley, 1995.

[52] T. Downing. *Java RMI: Remote Method Invocation.* IDG Books Worldwide, 1998.

[53] D. Duke, T. Green, and J. Pasko. *Research toward a heterogeneous networked computer cluster: The Distributed Queuing System version 3.0.* Supercomputer Computations Research Institute, Florida State University, 1994.

[54] R. Felderman, E. Schooler, and L. Kleinrock. The Benevolent Bandit Laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 1989.

[55] A. Ferrari. JPVM—network parallel computing in Java. In *Proceedings Workshop on Java for High-Performance Network Computing*, 1998.

[56] D. Flanagan. *JavaScript The Definitive Guide.* O'Reilly & Associates, Inc., 1997.

[57] I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[58] G. Fowler. The shell as a service. In *Proceedings of the Summer USENIX Conference*, 1993.

[59] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel virtual machine.* MIT Press, 1994.

[60] D. Gelernter and D. Kaminsky. Supercomputing out Recycled Garbage: Preliminary Experience with Piranha. In *Proceedings of ACM International Conference on Supercomputing*, 1992.

[61] GENIS Software GmbH. *CODINE 4.1.1, Technical Description.* Available at `http://www.genias.de/products/codine/tech_desc.html`.

[62] D. Ghormley, D. Petrou, S. Rodrigues, and A. Vahdat. GLUnix: A Global Layer Unix for a network of workstations. *Software Practice and Experience*, 1998.

[63] J. Gosling. The state of Java technology: Reality strikes! Kenote Speech, JavaOne Conference, 1998. Available at `http://java.sun.com/javaone/javaone98/keynotes/gosling/index.htm`.

[64] P. Gray and Sunderam V. IceT: Distributed computing and Java. In *Proceedings of the ACM Workshop on Java for Science and Engineering Computation*, 1997.

[65] A. Grimsaw and W. Wulf. Legion—view from 50,000 feet. In *Proceedings of the International Symposium on High-Performance Distributed Computing (HPDC)*, 1996.

[66] D. Griswold. The Java HotSpot virtual machine architecture. Available at `http://java.sun.com/products/hotspot/whitepaper.html`, 1998.

[67] W. Gropp, E. Lust, and A. Skjellum. *Using MPI: Portable parallel programming with the message-passing interface*. MIT Press, 1994.

[68] R. Henderson and D. Tweten. Portable Batch System. *NAS Scientific Computing Branch, NASA Ames Research Center*, 1995.

[69] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. Technical Report May, Oxford University Computing Laboratory, 1997.

[70] J. Hill and D. Skillicorn. Lessons learned from implementing BSP. In *High Perfomance Computing and Networking (HPCN'97)*. Springer-Verlag,, 1997.

[71] M. Hill, J. Larus, and D. Wood. Tempest: A substrate for portable parallel programs. In *Proceedings of the International Computer Conference (COMPCON)*, 1995.

[72] S Hirano. HORB: Distributed execution of Java programs, worldwide computing and its applications. In *Springer Lecture Notes in Computer Science*, 1997.

[73] S. Huang and Z. M. Kedem. Supporting a flexible parallel programming model on a network of workstations. In *Proceedings of International Conference on Distributed Computing Systems*, 1996.

[74] S. Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 1997.

[75] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring a method for scheduling parallel loops. *Communications of the ACM*, 1992.

[76] IBM. *IBM LoadLeveler: General information*, 1993.

[77] K. Jeong, D. Shasha, S. Talla, and P. Wyckoff. An approach to fault tolerant parallel processing on intermittently idle, heterogeneous workstations. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 1997.

[78] J. Jones and C. Brickell. Second evaluation of job queuing/scheduling software: Phase I report. Technical report, NAS Technical Report NAS-97-013, 1997.

[79] H. Karl. Bridging the gap between distributed shared memory and message passing. In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*, 1998.

[80] Z. M. Kedem, K. Palem, and P. Spirakis. Efficient Robust Parallel Computations. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 1990.

[81] D. Khandekar. QUARKS: Distributed shared memory as a building block for complex parallel and distributed systems. Master's thesis, Department of Computer Science, The University of Utah, March 1996.

[82] A. Kleine. Tya archive. Availabe at `http://www.dragon1.net/software/tya`.

[83] B. Kvande. The Java collaborator toolset, a collaborator platform for the Java environment. Master's thesis, Department of Computer Science, Old Dominion University, 1996.

[84] Valiant. L. A bridging model for parallel computation. *Communications of the ACM*, 1990.

[85] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transaction on Computers*, 1979.

[86] D. Lange, M. Oshima, G. Kargoth, and K. Kosaka. Aglets: Programming mobile agents in Java. *Lecture Notes in Computer Science*, 1997.

[87] E. Lazowska, L. Eager, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *Performance Evaluation Review*, 1988.

[88] J. Leon, A. Fisher, and P. Steenkiste. Fail-Safe PVM. In *Proceedings of the Workshop on Cluster Computing*, 1992.

[89] B. Li. *Free Parallel Data Mining*. PhD thesis, New York University, 1998.

[90] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings International Conference on Parallel Processing*, 1988.

[91] M. Litzkow. UNIX: Turning idle workstations into cycle servers. In *Proceedings the Summer USENIX Conference*, 1987.

[92] M. Litzkow, M. Livny, , and M. Mutka. Condor: A hunter of idle workstations. In *Proceedings International Conference on Distributed Computing Systems*, 1988.

[93] R. Lusk and R. Butler. Portable parallel programming with P4. In *Proceedings of the Workshop on Cluster Computing*, 1992.

[94] P. Madany. JavaOS: A standalone Java environment, a white paper, 1996. Available at `http://www.sun.com/javaos`.

[95] D. McLaughlin. *Scheduling Fault-tolerant, Parallel Computations in a Distributed Environment*. PhD thesis, Arizona State University, December 1997.

[96] D. McLaughlin, S. Sardesai, and P. Dasgupta. Calypso NT : Reliable, efficient parallel proces using on Windows NT networks. Arizona State University. Available at `http://calypso.eas.asu.edu`, 1997.

[97] D. McLaughlin, S. Sardesai, and P. Dasgupta. Preemptive scheduling for distributed systems. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1998.

[98] L. McMillan. An instructional ray-tracing renderer written for UNC COMP 136 fall '96. Available at `http://graphics.lcs.mit.edu/~capps/iap/class3/RayTracing/RayTrace.java`, 1996.

[99] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *Proceedings of the Network and Distributed System Security Symposium*, 1999.

[100] J. Moreira, V. Naik, and R. Konuru. A programming environment for dynamic resource allocation and data distribution. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1996.

[101] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. In *Performance Evaluation*, 1991.

[102] Network Wizards. *Internet Domain Survey, July 1998.* Available at `http://www.nw.com/zone/WWW/report.html`.

[103] C. Neuman and S. Rao. The Prospero resource manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 1994.

[104] D. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of SOSP*, 1987.

[105] ObjectSpace, Inc. *ObjectSpace Voyager Core Technology—The Agent ORB for Java*, 1998.

[106] Ohio Supercomputer Center, Ohio State University. *MPI primer/developing with LAM*, 1996.

[107] OpenMP Architecture Review Board. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, 1997.

[108] D. Patterson, D. Culler, and T. Anderson. A case for NOW (networks of workstations)—abstract. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1995.

[109] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. In *Proceeding of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1997.

[110] Platform Computing Corporation. *LSF User's and administrator's guide*, 1993.

[111] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. *Lecture Notes in Computer Science*, 1995.

[112] J. Pruyne and M. Livny. Interfacing condor and pvm to harness the cycles of workstation clusters. *Journal on Future Generations of Computer Systems*, 1996.

[113] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM Sigmetrics Conference on Mesurement and Modeling of Computer Systems*, 1993.

[114] RSA Data Security. *RSA Factoring Challenge.*

[115] RSA Data Security. *RSA Laboratories DES Challenge II.* Available at `http://www.rsa.com/rsalabs/des2`.

[116] RSA Data Security. *RSA Laboratories Secret-Key Challenge.* Available at `http://www.rsa.com/rsalabs/97challenge`.

[117] P. Salus. *Casting the Net: From Arpanet to Internet and Beyond.* Addison-Wesley, 1995.

[118] S. Sardesai. *Chime: A Versatile Distributed Parallel Processing Environment.* PhD thesis, Arizona State University, July 1997.

[119] S. Sardesai, D. McLaughlin, and P. Dasgupta. Distributed Cactus Stacks: Runtime stack-sharing support for distributed parallel programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[120] L. Sarmenta. Bayanihan: Web-based volunteer computing using Java. In *International Conference on World-Wide Computing and its Applications*, 1998.

[121] D. Scales and K. Gharachorloo. Towards transparent and efficient software

distributed shared memory. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, 1997.

[122] Jon Siegel. *CORBA Fundementals and Programming.* Wiley, 1997.

[123] Sun Micro Systems Inc. *Frequently Asked Questions - Java Security.* Available at `http://java.sun.com/sfaq`.

[124] Sun Micro Systems Inc. *JDK 1.2 Beta 2 Documentation.*

[125] Sun Micro Systems Inc. *Secure Computing with Java: Now and the Future.*

[126] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: a migratable parallel objects framework using Java. In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*, 1998.

[127] M. Theimer and K. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering*, 1989.

[128] K. Thitikamol and P. Keleher. Multi-threading and remote latency in software DSMs. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[129] University of Michigan. *The SLAPD and SLURPD Administrators Guide, Release 3.3*, 1996.

[130] L. Valiant. Bulk-synchronous parallel computers. In *Parallel Processing and Artificial Intellligence.* Wiley, 1989.

[131] R. Van Renesse, K. Birman, M. Hayden, and A. Vaysburd. Building adaptive systems using ensemble. *Software Practice and Experience*, 1998.

[132] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Ccmmunications of the ACM*, 1996.

[133] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1998.

[134] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol. RFC 1777, 1995.

[135] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251, 1997.

[136] G. Woltman. GIMPS discovers 37th. known Mersenne prime. Available at `http://www.mersenne.org/3021377.htm`, 1998.

[137] G. Woltman. Internet PrimetNet server. Available at `http://www.entropia.com/primenet/status_htm`, 1998.

[138] George Woltman. Great internet Mersenne prime search. Available at `http://www.mersenne.org/`.

[139] A. Yu, W. Cox. Java/DSM: A platform for heterogeneous computing. In *Proceedings of ACM Workshop on Java for Science and Engineering Computation*, 1997.

[140] M. Zekauskas, W. Sawdon, and B. Bershad. Software write detection for a distributed shared memory. In *Proceedings of USENIX Symposium on OSDI*, 1994.

[141] S. Zhou, J. Wang, X. Zheng, and P. Delisle. *Utopia: A load sharing facility for large, heterogeneous distributed computing systems.* Computer Systems Research Institute, University of Toronto, 1992.

Metacomputing on Commodity Computers

by

Arash Baratloo

Advisor: Zvi M. Kedem

External factors such as unpredictable behavior of computers and failures complicate the effective use of distributed multiuser environments as a parallel processing platform. This dissertation presents a unified set of techniques to build a metacomputer, that is, a reliable virtual shared-memory computer, on a set of unreliable computers. The techniques are specifically designed for adaptive execution of parallel programs on dynamic and faulty distributed environments. The dissertation presents four software systems to validate the feasibility of these techniques, both for networks of commodity workstations and for the World Wide Web, which lacks a shared file system and user-access control.

Calypso is a programming and a runtime system to address the difficulties of parallel programming on networks of workstations. The parallelism expressed by Calypso programs reflect the problem rather than the execution environment. The Calypso runtime system adapts computations to efficiently use the available resources: the number of workstations may grow and shrink dynamically, and the workstations may fail and slowdown at unpredictable times. *ResourceBroker* is a resource manager to facilitate the use of otherwise idle workstations. ResourceBroker demonstrates a novel technique to dynamically manage adaptive programs

that were not designed to work with external resource managers. As a result, a mix of adaptive programs, written using diverse programming systems can execute side-by-side on a set of workstations.

Charlotte leverages the code-mobility and secure execution of Java applets to extend the concept of metacomputing to the World Wide Web. Charlotte is the first parallel programming system to provide volunteer-based one-click computing: it allows any user on the Internet, without any administrative effort, to participate in ongoing computations by a simple click of the mouse. The Charlotte's runtime system transparently provides load-balancing and fault-masking. *KnittingFactory* is a software infrastructure that facilitates the execution of Charlotte programs. The contributions of KnittingFactory include a distributed directory service that migrates most of the computations to the client side, and a middleware service for direct applet-to-applet communication.