Charlotte: Metacomputing on the Web*

Arash Baratloo Mehmet Karaul Zvi Kedem Peter Wyckoff
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
New York, NY 10012, USA

Abstract

The World Wide Web has the potential of being used as an inexpensive and convenient metacomputing resource. This brings forward new challenges and invalidates many of the assumptions made in offering the same functionality for a network of workstations. We have designed and implemented *Charlotte* which goes beyond providing a set of features commonly used for a network of workstations: (1) a user can execute a parallel program on a machine she does not have an account on; (2) neither a shared file system nor a copy of the program on the local file system is required; (3) local hardware is protected from programs written by "strangers"; (4) any machine on the Web can join or leave any running computation, thus utilizing the dynamic resources.

Charlotte combines many complementary but isolated research efforts. It comprises a virtual machine model which isolates the program from the execution environment, and a runtime system which realizes this model on the Web. Load balancing and fault masking are provided by the runtime system transparent to the programmer. Charlotte provides distributed shared memory without relying on operating system or compiler support. It is implemented soley in Java without any native code, thus providing the same level of security, heterogeneity, and portability as Java.

In this paper, we describe the design and implementation of Charlotte and present initial performance results.

Keywords: Metacomputing, Distributed Computing, Parallel Programming Environments, World Wide Web.

1 Introduction

Over the last few years, the Internet has grown rapidly connecting millions of mostly idle machines. Its latest reincarnation as the World Wide Web has greatly increased its potential for utilization in diverse settings, including its potential to be used as a gigantic computing resource. On the other hand, utilization of local area networks as a parallel computing platform has been attractive for many years. There have been numerous research projects aimed at this goal. Based on their success, and as a natural evolutionary step, attempts have been made to extend existing systems from local area networks to wide area networks.

However, utilizing the Web as a metacomputing resource introduces new difficulties and problems different from those that exist in local area networks. First, many of the challenges that have been looked at individually (e.g., security, programmability, and scheduling) need to come together in a comprehensive manner. And second, the Web invalidates many of the assumptions used to build parallel environments for workstation clusters. For example, there is lack of a shared file system, no one has accounts on all connected machines, and it is not a homogeneous system. An environment to effectively utilize the Web as a metacomputing resource needs to address the following important issues: programmability, heterogeneity, portability, security, dynamic execution environment, and scalability.

Programming (Virtual Machine) Model: Generally, neither the programmer nor the end-user wants to deal with a dynamic and an unpredictable environment such as the Web. In fact, users are not interested in knowing whether their programs are running locally or remotely (perhaps in a distributed manner). For the Web to become an effective computing platform, the programming model needs to be decoupled from the dynamics of the execution environment. That is, programs need to be developed for a uniform and predictable virtual machine; the runtime system needs to realize the virtual machines. Otherwise, the task of program development becomes nearly intractable. We are not aware of any existing system providing a satisfactory solution for the Web.

Heterogeneity and Portability: The Web contains different types of hardware, running different operating systems, connected with different networks. Heterogeneity and portability are imperative to encompass the Web. Current heterogeneous systems are low level and generally employ a message-passing paradigm. High level systems

*This research was partially supported by NSF under grant numbers CCR-94-11590, CDA-94-21935, and GER-94-54173; and by AFOSR under grant number F49620-94-1-0132.

based on virtual shared memory, generally do not support heterogeneous environments.

Security and Accessibility: Cooperative work over the Web requires many facets of security measures. People need reassurance to allow "strangers" to execute computations on their machines. On local area networks this is accomplished by an administrator maintaining user access-rights and user accounts. Thus, the network becomes available only to a trusted set of users. This is not a feasible solution for the Web. In an ideal situation, any machine on the Web can contribute to any ongoing computation on the Web. We are not aware of any existing system that provides such a solution.

Dynamic Execution Environment: The Web is a dynamic environment comprised of many administrative domains; for example, machines become available and unavailable abruptly and network delays are unpredictable. Existing systems address these issues by providing some level of load balancing and fault masking. However, the extent of uncertainty is much greater on the Web.

A comprehensive solution for the Web requires that all of these issues be resolved. No system currently addresses all or even a majority of these. In this paper, we present a system called *Charlotte* that addresses these issues.

The research leading to our system started as a theoretical work where provable methods for executing parallel computations on abstract asynchronous processors were developed [11, 1]. The outline of the virtual machine interface to actual system was proposed in [10]. Theoretical results were then interpreted in the context of networks of workstations in [5]. The above were significantly extended and validated in the *Calypso* [2] system for homogeneous networks. Charlotte builds on these and other complementary research efforts by offering a unified programming and execution environment for the Web. Our work on Charlotte has resulted in several original contributions which are summarized below:

- Charlotte is the *first programming environment for parallel computing using a secure language*. The Java programming language guards against mischievous code attacking local resources. Charlotte is built on top of Java without relying on any native code. This means that a Charlotte program provides the same level of security and reassurance as a Java program.

- Charlotte is the first environment that allows *any machine on the Web to participate in any ongoing computation.* Two key factors make this possible. First, the system does not require a shared file system, nor does it require the program to reside on a local file system before a machine can participate in a computation. The Charlotte runtime system transmits the program to participating machines. Second, it is not necessary for a user to have an account (or any other type of privilege) to utilize a machine on the Web. The decision to involve a machine in a computation is made by the owner of that machine. These factors mean that potentially *any* machine can contribute to any running Charlotte computation on the Web.

- A novel technique for providing a *distributed shared memory abstraction without relying on operating system or compiler support*. Current techniques for shared memory require either support from the operating system (in the form of setting page access-rights) or a compiler (to generate the necessary run-time code at each memory access). Shared memory systems based on operating system support are neither system independent nor safe. Compiler-based shared memory systems are tied to a particular programming language and a particular target language. Our approach does not suffer from these limitations, which makes it possible to provide virtual shared memory at the programming language level.

- We extend some of our previous work originally developed in a different setting to deal with the dynamics of the Web. Two integrated techniques, *eager scheduling* and *two-phase idempotent execution strategy* are used for load balancing and fault tolerance.

- We leverage existing isolated contributions by providing a *unified and comprehensive solution*. The Java programming language is used for heterogeneity and portability. Our programming environment can be conceptually divided into a virtual machine model and a run-time system. The virtual machine model provides a reliable shared memory machine to the programmer and isolates the program from the execution environment. The run-time system realizes this model on a set of unpredictable, dynamically changing, and faulty machines.

2 Related Work

PVM [17] and MPI [8] are representatives of message passing systems. They provide portability and good performance, but they are low level. Systems such as CARMI [15] and [9] augment PVM's functionality by providing resource management services. However, they are limited to local area networks: there is no support for dynamic load-balancing and fault masking, they require the program to reside at each site (or a shared file system), and they require the user or the system to have an account on each machine participating in the computation. These factors severely limit their use as an interface to a metacomputing framework on the Web.

Another class of systems for distributed computing focuses on providing distributed shared memory (DSM) across loosely-coupled machines. IVY [13] and TreadMarks [12] are representatives of DSM systems. Cilk [3] is a comprehensive system providing resource management and fault-tolerance in addition to DSM. However, it makes similar assumptions about the file system and user privileges as message passing systems, which again, limits its applicability to the Web. In addition, DSM systems in general, do not work on heterogeneous environments.

Recently, with the introduction of Java, another class of systems is becoming available. HORB [16] and the E programming language [7] are two such examples. HORB is a distributed Object Oriented system. It extends Java with a well understood programming model, RPC, and persistent objects. Charlotte and HORB are similar in that they both utilize Java in providing heterogeneity, interoperability, and security. However, we provide a virtual machine model, shared memory abstractions, load-balancing and fault-masking.

The E programming language extends Java by adding message passing through channels, and richer security through cryptographic and authentication. We do not address these issues. For example, we do not authenticate the correctness of results: malicious sites could make themselves available and report wrong answers. However, Charlotte's design is not tied to any particular language or system, making it possible to leverage systems such as E.

3 Charlotte's Programming Model

If a parallel program is to utilize the Web, its execution environment is not known at development time—the number of available machines, their location, their capabilities, and the network cannot be predicted ahead of time. In general, a program's execution environment will differ for each invocation. To deal with the dynamics of the execution environment, either the programmer must explicitly write adaptive programs, or a software environment, such as a runtime system, must deal with the dynamics. We feel that the former solution puts too much strain on the programming effort. We conjecture that for an effective utilization of the Web, the programming model must be decoupled from the execution environment. Programs should be developed for a uniform and predictable virtual machine, thus, simplifying the task of program development; the runtime system should implement virtual machines and deal with the dynamics.

Charlotte allows high level programming based on the parallelism of the problem and independent of the executing environment. Programs are written for a virtual parallel machine with infinitely many processors sharing a common name-space. Integrating a machine into a running computation, balancing the loads among different machines, detecting and removing failed machines from a computation, and maintaining the coherence of distributed data are transparently provided by the runtime system.

```
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() {
     ...
  }

  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() {
     ...
     parBegin();
     addDroutine(this, Size);
     parEnd();
     ...
  }
}
```

Figure 1: Matrix multiplication in Charlotte.

3.1 Parallel Steps and Concurrent Routines

An execution of a Charlotte program consists of alternating sequential and parallel steps. A sequential step consists of standard sequential Java code. Computationally intensive parts of a program are done in parallel steps by one or more *routines*. A routine is analogous to a standard thread in Java, except for its capability to execute remotely. A parallel step starts with the keyword (which is in reality a function call) `parBegin` and ends with `parEnd`. There can be any number of routines defined within one parallel step. The first argument to `addDroutine` is a subclass of `Droutine` implementing `drun()`. `drun()` takes two arguments: the number of concurrent routines and the routine identifier (in the range $0,\ldots,$`numTasks`-1). The second parameter of `addDroutine` defines the number of instances to run concurrently (which is passed to `drun()`). The closing `parEnd` serves as a synchronization barrier.

Figure 1 shows a Charlotte program which multiplies two square matrices using 500 routines. Each routine is responsible for producing one row of the resultant matrix. The important points to note here are:

- The program's parallelism is based on the parallelism of the problem at hand and not the execution environment. The same program can execute on one or any number of machines depending on availability.

- The programmer need not be aware of the fact that this application can be executed in a distributed fashion, utilizing multiple machines on the Web.
- Integrating machines into the computation, load-balancing, fault-masking, and data coherence are transparent.
- The services provided by Charlotte do not require any language or compiler modifications.

3.2 Distributed Shared Memory Semantics

In a Charlotte program, the data is logically partitioned into *private* and *shared* segments. Private data refers to what is local to a routine and cannot be seen by others. The shared data is distributed and can be seen by others. For reasons to be discussed in Section 5, we have chosen to implement distributed shared memory within the language, at the data type level. That is, for every basic data type in Java, there is a corresponding Charlotte data type implementing its distributed version. For example, where Java provides `int` and `float` types, Charlotte provides `Dint` and `Dfloat` types (classes). Since Java prohibits operator overloading, distributed objects are accessed and modified with explicit function calls (*get()* and *set()*). The consistency and coherence of the distributed data is maintained by the runtime system.

In an attempt to improve performance, many DSM systems have introduced multiple memory-consistency semantics [14]. The decision as to which consistency model is best suited for a particular application is left to the programmer. We feel this generally complicates a task the researchers were seeking to simplify. In Charlotte, we provide a single and intuitive memory semantics: *Concurrent Read and Exclusive Write* (CR&EW). This means that one or more routines can read a data variable, and at most one can modify its value. More specifically, a read operation returns the value of the object at the time the parallel step began, and a write operation becomes visible at the completion of the parallel step.

The CR&EW semantics are implemented with the *Atomic Update* protocol. Intuitively, this means that write operations are performed locally and propagated at the completion of the routine. The advantages of this protocol are that (a) network traffic is reduced by packing many modifications into one network packet; (b) write operations are efficient since they are performed locally and do not require invalidation; (c) atomic execution of routines is guaranteed (each one is executed in an all-or-nothing fashion); (d) each routine can execute in isolation and can be verified independently of the execution order; (e) it simplifies the control logic in the coherence protocol. It should also be noted that the atomicity of updates is fundamental for our load balancing and fault masking techniques, which will be discussed later.

4 Runtime System

The Charlotte package provides three major services: (1) scheduling service, (2) memory service, and (3) computing service. In the current prototype, the scheduling service, the memory service, and parts of the computing service executing the sequential steps are fused into a single process—the *management service*, or the *manager* for short. The computing service performs the computationally intensive parallel routines. We refer to the processes that perform the computing service as *workers*. A single manager and one or more workers together implement the virtual machine model that the program was written for.

We assume that a user has one machine under her control and that this machine is highly reliable. The user, however, wishes to utilize other machines on the Web which may be unreliable. We will sketch the overall execution strategy here.

When a user starts executing a Charlotte program, it logically runs as two separate processes: one performing as the manager, the other performing as a worker. The manager executes the program until it reaches a parallel step. Upon reaching this point, it calculates the number of instances for each `addDroutine` and creates a *Dispatch Table* for bookkeeping. We will refer to each instance of a routine as a *job*. The manager then waits (if necessary) for workers to contact it. In general, this will happen after a worker finishes a previous assignment and is willing to work. However, a worker could just appear any time and be integrated into the computation. In general, the manager keeps giving jobs to workers until they are done. The association of workers and managers will be described in Section 4.2. We will first describe the load-balancing and fault-masking techniques used.

4.1 Load Balancing and Fault Masking

As the manager assigns jobs to workers, it has the option of assigning a job repeatedly until it is executed to completion by at least one worker. This technique is called *eager scheduling*. Obviously, eager scheduling does not provide correct program behavior without a suitable memory management scheme—as the same program fragment could get executed multiple times. However, it should be clear that given an appropriate memory management scheme, eager scheduling alone has the following properties:

- As long as at least one worker does not continuously fail, all jobs will be completed.
- New workers can be integrated into a computation anytime, even in the middle of a parallel step.
- A crash-failed machine or a slow machine running a worker process is transparently bypassed by faster workers, providing fault tolerance.

- Worker processes running on slow machines ask for jobs less frequently, and thus, do less work. This results in load balancing that is automatic and transparent to both the programmer and the user.

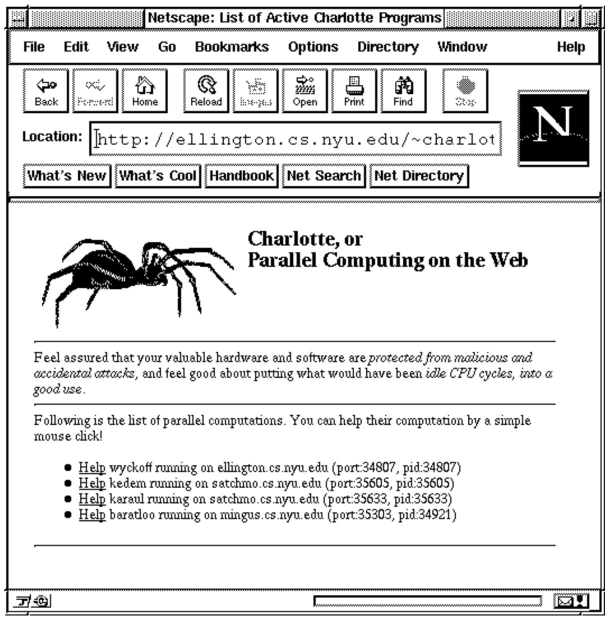
A simplistic application of eager scheduling does not produce a correct program behavior since a single job may be executed several times. Problems to be addressed include inconsistent memory views across jobs and even across various "copies" of the same job, violation of exactly-once semantics by multiple executions, and excessive data traffic. The memory management technique that we employ is called *two-phase idempotent execution strategy* (TIES). TIES guarantees correct execution of shared memory programs under eager scheduling.

TIES works as follows. Before the execution of a job, a worker process invalidates its distributed data types. As it executes, the read data is demand-paged from the manager, however, the writes occur locally in the workers memory space. It is at the termination of a job that the dirty data is sent back to the manager, indicating its completion. The manager buffers the incoming data, keeping only one copy of the modifications for each job (despite potential duplicates). It discards all other responses, including those from slow workers that have completed an old job. The first phase of TIES ends when at least one invocation of each job reports back. The second phase consists of the manager applying the modifications to its memory space, completing the execution of the parallel step.

4.2 Association of Workers and Computations

It is very likely that at any moment there are idle machines on the Web willing to help in some computation, and there are computations that could utilize them. Both sets are dynamic. The first difficulty lies in associating an idle machine with a computation. It is only when this association has been established, and the necessary program fragment been made accessible, that an idle machine can contribute to the computation. However, the lack of trust in allowing "strange programs" to execute on local hardware is a serious issue. The complete problem is of great importance since it lies at the core of providing an effective metacomputing environment.

We have adopted a solution that does not scale for settings such as the World Wide Web, but it is an effective solution for our network at New York University. Oversimplifying it, when a Charlotte program reaches a parallel step, it registers itself with a specific daemon process. This action creates an entry in a URL homepage (see Fig. 2). Any user in our network can visit this homepage using any Java-capable browser and see the list of active programs. If the user wishes to donate some of the CPU power of her machine, she can simply click on the entry. This will load the required code to the user's machine and start assisting the ongoing computation. We plan to automate this feature by relying on manager lookup services.


Netscape: List of Active Charlotte Programs
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Location: http://ellington.cs.nyu.edu/~charlot
What's New What's Cool Handbook Net Search Net Directory

Charlotte, or
Parallel Computing on the Web

Feel assured that your valuable hardware and software are *protected from malicious and accidental attacks*, and feel good about putting what would have been *idle CPU cycles, into a good use*.

Following is the list of parallel computations. You can help their computation by a simple mouse click!

- Help wyckoff running on ellington.cs.nyu.edu (port:34807, pid:34807)
- Help kedem running on satchmo.cs.nyu.edu (port:35605, pid:35605)
- Help karaul running on satchmo.cs.nyu.edu (port:35633, pid:35633)
- Help baratloo running on mingus.cs.nyu.edu (port:35303, pid:34921)

Figure 2: Sample list of active Charlotte programs.

Since Charlotte is entirely implemented in Java, it provides the same security guarantees as Java. Java guarantees the protection of local resources from programs. Although there seem to be security holes in the current implementation [6], there are strong indications that this will be solved. Charlotte will transparently take advantage of these improvements. Once users stop being afraid of programs that come over the network, we feel that they will have more incentive to allow others to use their idle CPU cycles

It is important to stress that Charlotte supports heterogeneous systems, which makes it possible for any idle machine to execute a parallel computation along side any other—a necessity for the Web.

5 A Novel Technique for DSM

To provide the abstraction of a single address space to multiple programs running on different computers, DSM systems must detect accesses to shared memory and propagate updates. For a shared memory system to be realized over the Web, hardware and operating system independence is a necessity, and language/compiler independence is desirable. In Charlotte, we achieve both.

5.1 Existing Techniques

There have been two approaches to implement DSM at the software level: one relies on virtual memory page protection and the other on a compiler to provide software write detection.

Traditional software-based shared memory systems rely on virtual memory page protection [13, 2, 12]. Detection of a data access is done by protecting the memory pages and catching the page-fault signal generated by the operating system. A write operation sets a dirty-bit for each page, indicating that the change needs to be propagated. The granularity of shared data segments is determined by the system—a virtual memory page. This can result in *false sharing* when the same memory page is accessed independently by multiple processes. Since page size and data format vary across different machines and operating systems, this is not a viable option for a heterogeneous environment such as the Web.

Another technique to implement DSM relies on compiler and runtime support [18]. A compiler inserts the necessary code to detect and service an access to the shared memory region. This approach has the advantage that the granularity can be controlled, meaning that it alleviates false sharing. On the other hand, it requires that the system continuously evolves with the language it is based on.

Charlotte's shared memory abstraction is neither operating system nor compiler based. We were introduced to the feasability of software write detection by the work in [18]. However, rather than using a compiler, we chose to realize shared memory abstraction through shared name-space. This method combines the advantages of both previous methods:

- Programs can run on any combination of hardware and operating systems where Java is available.
- Charlotte does not need to evolve with every change to the Java language and compilers.
- The abstraction of shared memory can be provided in a heterogeneous environment.
- The granularity of shared data can be controlled dynamically.

5.2 Prototype Implementation

Charlotte's distributed shared memory is implemented within the language, at the data type level; that is, through Java classes. In addition to a `value` field, each data object maintains its state which can be one of `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 that can be used in a read operation; and `dirty` indicates that the local value is correct and that it has been modified.

Distributed objects are read and written through class member functions. A read operation on a `readable` or a `dirty` object returns its `value`. Otherwise, its value is retrieved from one of the manager processes over the network, and then changes state to `readable`—this corresponds to demand-paging. On a write operation, the `value` is modified and the object changes state to `dirty`—this corresponds to setting the dirty bit. A `dirty` object propagates its value at the end of the job.

The problem in this implementation comes up when an instance of an object in `not_valid` state is accessed. How does *this particular instance* convey its identity to the manager to retrieve its value? After all, the manager has many instances of the same class. Our solution is based on a unique identifier, assigned deterministically based on instantiation order. This requires that distributed objects be instantiated in the same order at each site. For simplicity, we chose to instantiate all the distributed objects in the constructor of a single class. This guarantees that each replica of a distributed object is assigned the same identifier.

We have implemented two techniques to improve the performance of shared memory in Charlotte. First, a read operation does not result in transferring the value of a single item. Instead, multiple items are shipped in a single network packet. The size of each packet can be set dynamically at runtime, thus adjusting the granularity of shared memory. Second, the information as to which objects are invalidated is piggy-backed with the manager's job assignment. Thus, each worker page-faults on read-only data items at most once. Our general implementation is not particularly efficient, but simple and tractable.

In summary, our approach for realizing distributed shared memory on loosely coupled machines does not rely on virtual pages (operating system) nor a compiler. Similar to compiler based systems, it runs in the user-space, supports variable data granularity, and avoids false sharing. On the other hand, it is not dependent on any particular implementation of the language or compiler. It is important to note that this technique is not tied to any particular memory coherence model—this is a general technique that can be used to implement many different memory coherence models. For example, both sequential and release consistency memory models can be implemented using our technique.

6 Experiments

Here we present initial performance results. Experiments were conducted using up to 10 Sun SPARCstation 5 machines connected by a 10Mbit Ethernet. Reported times are wall clock times, and hence account for *all overheads*. Speedups are with respect to a sequential execution of a standard Java program. The same executable was tested in each case: the program and the runtime parameters did not change.

A C program runs an order of magnitude faster than a Java program. However, Java compilers, due to be released later this year, will provide performance much closer to C. We expect our results to carry over.

We chose a scientific application from statistical physics—computing the 3D Ising model [4]. This is a simplified model of magnets on a three dimensional lattice which can be used to describe qualitatively how small systems behave. Computing the Ising model involves an exponential number of independent tasks and very little data

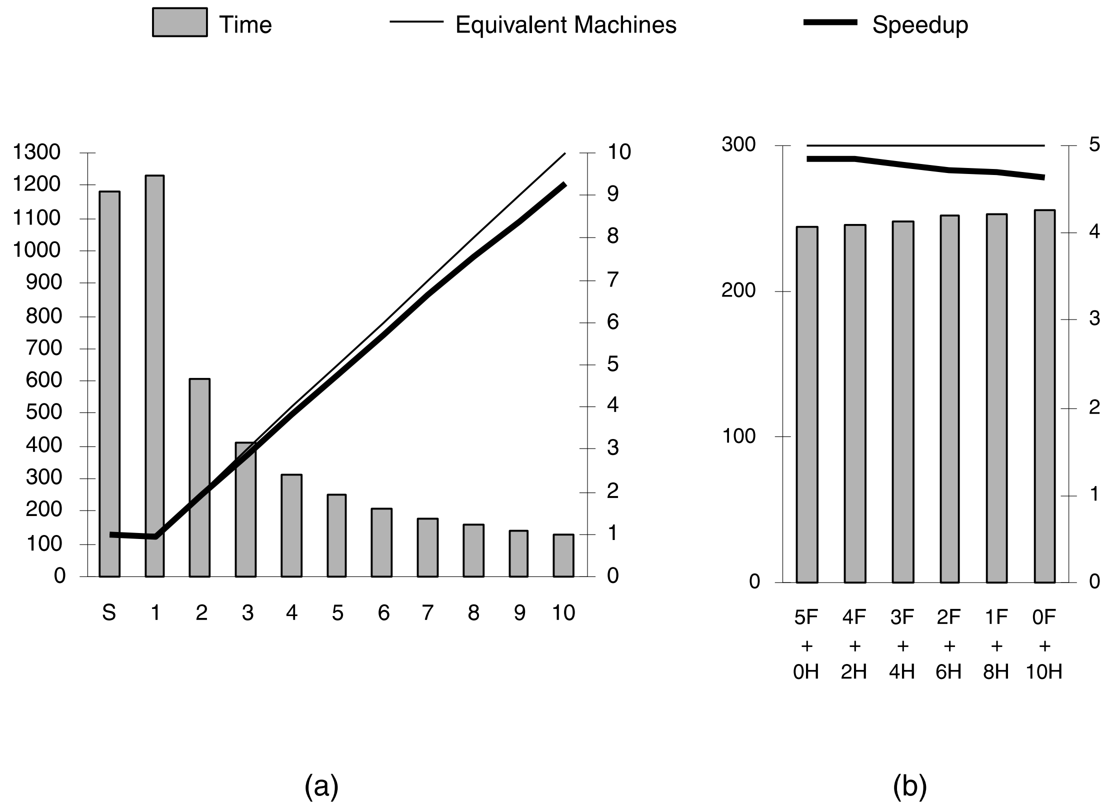


Figure 3: Performance of a Charlotte program running the Ising model. The x-axis denotes the individual experiments. In part (a), **S** represents the sequential execution, and the others represent the execution of a Charlotte program running on 1 to 10 machines. In Part (b), **F** (full) and **H** (half) represent a machine which contributes 100% and 50% of its CPU cycle, respectively. The left y-axis denotes the time. The right y-axis denotes equivalent machines and the speedup.

movement.

We performed three experiments. First, we study the performance and overhead of Charlotte. Second, We examine the utilization of slow machines in a computation. In particular, we are interested to see whether the addition of slow machines will affect the overall performance. Third, we analyze how well the system can integrate machines into an ongoing computation, and how efficiently failures can be masked.

A sequential Java program to compute the Ising model, with a period of 23, ran in 1,186 seconds. The equivalent Charlotte program ran in 1,230 seconds on one machine (the manager process and one worker process ran on the same machine). This results in 96% efficiency. The same program on two machines (one machine running the manager and one worker, and the second machine running the second worker) ran in 609 seconds. This represents a speedup of 1.95 and 97% efficiency compared with the sequential Java program. Fig. 3 (a) shows the performance for 1 through 10 workers. Given the high level programming model and a need for optimization, we were gratified that the program achieves 93% efficiency with 10 machines. This is competitive with other systems that do not provide load balancing and fault masking which Charlotte does.

In the second set of experiments we evaluate how efficiently Charlotte can handle an environment composed of some fast and some slow machines. This models a "real" setting on the Web. We used machines that were available 100% and machines that were available 50% of the time. We performed six tests. In all tests, the number of actual machines varied from 5 to 10, although the effective number of machines was always 5. For example, we ran a test with 3 machines that were available 100% of the time and 4 machines that were available 50% of the time. As Fig. 3 (b) indicates, Charlotte's load balancing technique is effective in this environment.

In the final experiment we measure how effectively a machine can be integrated into a running computation and how well Charlotte handles failures. In this test, we ran the program on five machines (one machine running the manager and one worker, and the other four machines running workers only). After 100 seconds, one worker was crashed and instantly a new worker started. After another 100 seconds, two workers were crashed and instantly two new workers were started. This program completed in 275 seconds, as opposed to 248 seconds in a perfect setting. This indicates that Charlotte performs well in a dynamic situation.

7 Conclusion

Charlotte supports some of the key functionality critical for harnessing the Web as a metacomputing resource for parallel computations.

It provides programmers with a convenient and stable virtual machine interface to the heterogeneous and unpredictably behaving execution environment. The programming model is based on shared memory and employs the emerging Java standard enhanced with a few classes for expressing parallelism. Thus, heterogeneity and security are provided to the extent supported in Java. Furthermore, the runtime environment realizes automatic load balancing and fault tolerance, both critical to the effective utilization of the Web. No other system provides this gamut of functionality for parallel computing in a wide area network of heterogeneous machines.

Although Charlotte is currently handicapped by the slow performance of interpreted Java, this will be transparently resolved once Java compilers are released.

References

[1] Y. Aumann, Z. Kedem, K. Palem, and M. Rabin. Highly efficient asynchronous execution of large-grained parallel programs. In *Proc. of 34th IEEE Annual Symposium on Foundations of Computer Science*, 1993.

[2] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proc. of IEEE International Symposium on High-Performance Distributed Computing*, 1995.

[3] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, A. Shaw, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. Symposium on Principals and Practice of Parallel Programming*, 1995.

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

[5] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: A fault-tolerant, high performance approach. In *Proc. of the 15th International Conference on Distributed Computing Systems*, 1995.

[6] D. Dean, E. Felten, and D. Wallach. Java Security: From HotJava to Netscape and Beyond. To appear in *Proc. IEEE Symposium on Security and Privacy*, 1996.

[7] Electric Communities. *The E Programming language*. Available at http://www.communities.com/e/epl.html.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing-Interface*. MIT Press, 1994.

[9] J. Ju and Y. Wang. Scheduling PVM Tasks. *Operating Systems Review*, July 1996.

[10] Z. Kedem and K. Palem. Transformations for the automatic dirivation of resilient parallel programs. In *Proc. of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1992.

[11] Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. of 22nd ACM Symposium on Theory of Computing*, 1990.

[12] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. *In Proc. of the Winter USENIX Conference*, 1991.

[13] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. *In Proc. International Conference on Parallel Processing*, 1988.

[14] D. Mosberger. Memory Consistency Models. Technical Report number TR92/11, University of Arizona, 1992.

[15] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. *In Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1995.

[16] H. Satoshi. *The Magic Carpet for Network Computing: HORB Flyer's Guide.* Available at http://ring.etl.go.jp/openlab/horb/doc/guide/guide.html.

[17] V. Sunderam, G. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 1994.

[18] M. Zekauska, W. Sawdon, and B. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proc. of Symposium on OSDI*, 1994.