ParaWeb: Towards World-Wide Supercomputing †

Tim Brecht, Harjinder Sandhu, Meijuan Shan, and Jimmy Talbot
Department of Computer Science
York University, North York, Ontario
{brecht,hsandhu,shan,talbot}@cs.yorku.ca

September 19, 1996

Abstract

In this paper, we describe the design of a system, called ParaWeb, for utilizing Internet or intra-net computing resources in a seamless fashion. The goal is to allow users to execute serial programs on faster compute servers or parallel programs on a variety of possibly heterogeneous hosts. ParaWeb provides extensions to the Java programming environment (through a parallel class library) and the Java runtime system that allow programmers to develop new Java applications with parallelism in mind, or to execute existing Java applications written using Java's multithreading facilities in parallel. Some experimental results from our prototype implementation are used to demonstrate the potential of this approach.

1 Introduction

The recent explosion of the World-Wide Web has enabled a diverse set of new Internet related applications and technologies. Among the key attractions of the Web has been the promise of easy access to a wealth of information around the world. Currently, however, the Web supports only a simple model of computing: information residing at a host computer can be automatically downloaded and viewed on a client machine, and applications residing at a host computer can be automatically downloaded and executed on the client machine. This model, although extremely powerful, exploits only a fraction of the potential of the Internet. A more compelling vision is one that views the Internet as a source of both information and computing resources. In the ParaWeb project, we are building tools and abstractions that allow users to treat the entire Internet as a single large computing resource. Within the ParaWeb framework, users may execute serial programs remotely, on faster compute engines, or they may execute parallel programs on a variety of platforms across the Internet.

The mechanisms for ParaWeb are partially available today in the form of the Java programming language [5] and Java Virtual Machine (JVM) [7]. Java applications are compiled into an intermediate byte-code that can be executed (interpreted) on any architecture, relieving both the developer and user of concerns related to the heterogeneity of the target platforms. While the interpreted nature of the byte-code makes the current execution very slow compared to equivalent programs written in C or C++, future just-in-time compilers will likely eliminate much of this performance gap.

The rapid proliferation of the Web has been due to seamless access it provides to information that is distributed across an organization or around the world. The key to the ParaWeb model is to build a software infrastructure that provides the same level of seamless access to heterogeneous computing resources, either within an organization or across the entire Internet. In addition to allowing existing unmodified thread-based Java applications to be automatically executed remotely or in parallel, ParaWeb introduce a new class library which allow new Java applications to be written specifically with parallelism in mind.

A version of this paper appears in the **Proceedings of the Seventh ACM SIGOPS European Workshop**, Connemara, Ireland, September, 1996, pp. 181-188.

The ParaWeb model is described in Section 2 while the mechanisms used to implement ParaWeb are described in Section 3. In Section 4 we present early experiences with a simple parallel application. Section 5 outlines a number of issues being explored and addressed in this work and Section 6 describes related work. We conclude the paper with a summary, in Section 7.

2 ParaWeb

Currently, utilizing remote computing resources is a complex and time consuming task. Many national laboratories, for example, make computing time available on powerful supercomputers that are too expensive for the average user to purchase. But in order to use one of these machines, the user must apply for and obtain an account, log in, write, compile and debug the program, and then execute the program and manually collect the results. Furthermore any program written for that machine must be ported, recompiled, and debugged prior to execution on another type of machine.

Similarly, effectively utilizing computing resources within an intra-net is equally challenging for the average user. While load sharing facilities, such as Utopia [11], and parallel programming libraries such as PVM [4], provide some support for running applications on heterogeneous architectures, the user must still compile, debug, and install the application on each platform on which it is to run. As as result, utilizing heterogeneous computing resources is rarely done in practice.

By leveraging its key mechanisms on existing Java technology, ParaWeb programmers need not worry about issues related to heterogeneity. ParaWeb provides mechanisms for the execution of Java programs remotely or in parallel in a simple yet elegant way. Using Java, programmers write, debug and compile (to byte-code) programs on their own machine. The same byte-code can be executed on any server, with each server emulating a strictly defined virtual machine. Consequently, the programmer need never worry about problems such as data alignment, data representation and operating system compatibilities, that are typically associated with heterogeneous distributed or parallel computing.

ParaWeb extends the conventional Web model in a number of ways. First, ParaWeb permits clients to download and execute a single Java application in parallel, on a network of workstations. Second, clients can automatically upload and execute programs on remote compute servers. In the latter case, a site willing to act as a compute server, such as the national laboratory in the previous example, or even compute servers within an organization, run byte-code interpreters which in turn execute code on behalf of the client. The program is automatically uploaded and executed on a compute server and results are returned to the client. In the case of a parallel application, the client may upload code to many heterogeneous compute servers within an organization or even throughout the Internet.

3 ParaWeb Implementation

In ParaWeb, we are designing, building and experimenting with mechanisms for parallel computation within the Java framework. This framework, consisting of the Java programming system and the Java runtime system, may be extended in a variety of ways to achieve parallelism. The obvious approach is to extend both the Java programming system and the runtime system to provide a complete parallel programming and computing environment. However, it is possible to achieve parallelism in Java by extending the programming system (through a set of parallel classes) without modifying the Java runtime system. Further, since Java already provides some basic mechanisms for concurrency, including threads and synchronization, it is also possible to modify the Java runtime system to achieve parallelism without extending or modifying the Java programming environment.

The suitability of each of these approaches to parallelism in the Java framework depends on the nature of the target parallel computing environment. In ParaWeb, we are building two distinct sets of mechanisms based on the latter two approaches, the first called the Java Parallel Runtime System (JPRS), and the second called the Java Parallel Class Library (JPCL). Our decision to support these as two separate mechanisms

for parallel computing is based upon the observation that parallel computing on the web will likely take two predominant forms:

1. people will wish to run third-party Java applications faster. This may mean simply sending the application to a faster compute engine for execution, or it may mean sending a single application that is capable of running in parallel to a variety of compute engines, either local or remote. Since no assumption can be made about the mechanisms for parallelism used by these third-party applications, the most suitable approach is one that extends the Java runtime system but makes no assumptions about the program (other than that it uses the basic facilities for concurrency that Java already provides). This would allow existing Java applications that use only standard Java concurrency facilities to be executed in parallel.

2. people will wish to write applications in Java that exploit parallelism, and distribute these applications to other users. In this case, the most suitable approach is one which extends the programming environment but not the runtime system. This would allow Java programs written explicitly for parallelism using the parallel programming environment to run in parallel on existing unmodified Java interpreters.

A secondary distinction in the current implementation of these two approaches is that the former (the JPRS) supports a distributed shared memory framework while the latter (the JPCL) supports message passing (this distinction may disappear as the classes in the JPCL and those provided in future Java distributions both continue to evolve). Because Java provides a threads abstraction, executing standard thread-based programs across multiple platforms on the JPRS requires presenting these threads with the illusion of shared memory, so that from the perspective of the program, the underlying system is no different from the standard system in which all threads execute on the same machine. On the other hand, providing communication mechanisms between remote threads supported only by the JPCL class library (executing on the standard Java runtime system) is easiest to do using a message passing framework. In the following subsections, we first describe the implementation of the Java parallel class library, and the mechanisms for uploading applications or threads to remote servers for execution within this approach. We then describe the parallel runtime system approach.

3.1 The Java Parallel Class Library (JPCL)

The parallel class library in ParaWeb provides mechanisms for the remote creation and execution of threads, and facilitate communication between these threads. A programmer can create a thread that executes on a remote machine by simply extending the JPCL RemoteThread class and then instantiating the new class. Communication is currently provided in the form of standard message passing *send* and *receive* primitives.

The sequence of steps performed in order to execute a parallel program in our current implementation are labeled in Figure 1 and described below.

1. Server sites (Server A, Server B and Server C in Figure 1) start the Java interpreter and initially run a daemon which has been compiled to Java byte-code. Each server daemon then registers with a local scheduling server which is used to keep track of which servers daemons are available, the periods of time during which they are available, authentication and authorization policies, server daemon loads and any other information that is useful for determining which server daemons to utilize and when. At this point these server daemons are ready to accept and execute Java byte-code on behalf of the permitted clients.

2. A Java program creates a thread on a remote machine by simply instantiating a new object that has extended the RemoteThread class. When this object is instantiated on the client machine, the library code implementing the RemoteThread class contacts the scheduling server and requests the address of a server daemon that will execute code on behalf of the client (provided a machine name has not been specified for the RemoteThread).

3. The scheduling server replies with an available server daemon. If a number of remote threads are going to be created, the scheduling server can be asked for a specified number of server daemons. This may also be necessary so that the threads being executed on the remote servers will be able to communicate if the need arises.

4. The instantiated object on the client sends the compiled byte-code (class file) to the remote server daemon, which uses our Java network class loader to load the class from the network directly into memory (rather than copying it to disk and then loading it to memory). The server daemon then calls the run method of the uploaded RemoteThread class which starts the execution of the thread. In the example in Figure 1 the scheduling server has determined that Server A and Server B should be used, while Server C is not used in the execution of this parallel program.

5. The server daemon executes the client's code and sends results back to the client as required by and specified in the application.

6. When the client application completes, it contacts the scheduling server to inform it that it has finished using the server daemons. Note that the server daemons may also be involved in notifying the scheduling server when they have completed the execution of a client's thread. However, the client may wish to further utilize the server to execute other remote threads before its application is completed.

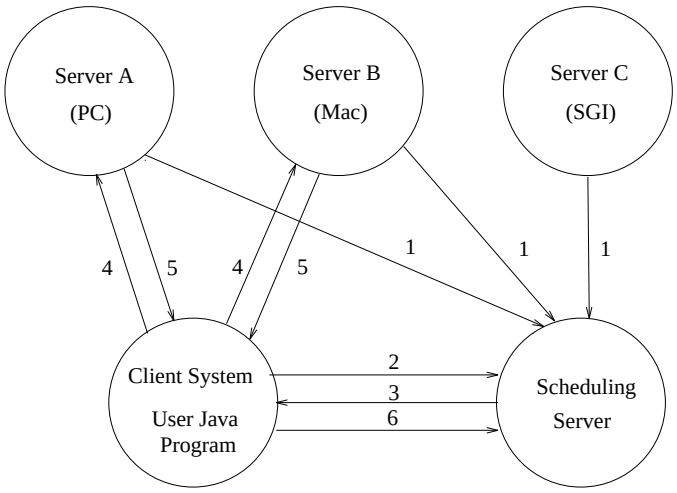

Figure 1: Running a parallel program using the Java parallel class library.

One of the major advantages of this approach is that the server daemons execute standard Java byte-code within an unmodified interpreter. Therefore, the server daemon is completely portable and capable of being executed on any computer which executes Java byte-code. The daemon could, for instance, be run inside a Java capable browser.

Naturally, in order to deploy this type of system in a large scale environment, the scheduling server will play a central role. We are developing scheduling servers which will be used to keep track of servers daemons within a local domain as well as methods of communicating with schedulers in other domains. Schedulers will communicate with other schedulers in different domains when sufficient resources are not available locally. These schedulers will also be written using Java so they can be executed portably on a number of platforms.

3.2 The Java Parallel Runtime System (JPRS)

In the implementation of the ParaWeb JPRS, we modify the Java interpreter to allow threads to be instantiated on any remote machine that is running the modified interpreter. The Java interpreters on each machine coordinate with each other to maintain the illusion of a global shared address space for the application they are running, in much the same way as distributed shared memory systems [9][6][1] maintain the illusion of shared memory. A crucial element in this approach is that the definition of the Java Virtual Machine remains unchanged, as does the byte-code of existing Java programs, so that no new libraries or annotations need be added to existing Java programs. Thus, any compiled Java application utilizing threads may be executed in parallel, by distributing those threads to different machines. Clearly, one would only want to distribute compute intensive threads, but determining *a priori* which threads are compute intensive is a non-trivial task.

Our implementation of the Java parallel runtime system relies on the existing Java mechanisms for concurrency and synchronization. In Java, a new thread is created upon the instantiation of an object that extends the *Thread* class. Threads may share any object to which they have a reference. Synchronization is achieved using a monitor-like facility. The keyword *synchronized* placed in front of a method definition implies that any thread executing that method must gain exclusive access rights prior to executing the method. Within a synchronized method, a thread may call *wait*, to temporarily halt execution of the thread and allowing another thread to execute a synchronized method in that class. The original thread resumes execution only when another thread calls *signal*.

Figure 2 shows an example of a parallel matrix multiplication using Java threads. A series of threads are created, one per row of the result matrix. Each of the threads is passed a reference to the two input matrices and the result matrix, and each thread in turn calls a method to multiply the row it has been assigned (the constructor, which accepts the parameters, is called when the class is instantiated, and the *run* method in the class is executed when the *start* method is called on that class).

```
class MatrixMultiply {
   public void multiply(matrix A, matrix B, matrix C) {
      for (int i=0; i<A.nrows(); i++)
          (new VectorMultiply(i, A, B, C)).start();
   }
}

class VectorMultiply extends Thread {
   matrix A, B, C;
   int indx;
   VectorMultiply(int i, matrix X, matrix Y, matrix Z) {
       A = X; B = Y; C = Z; indx = i;
   }
   public void run() {
       C.vector_multiply(indx, A, B);
   }
}
```

Figure 2: Example Matrix Multiply code

In the JPRS, upon the invocation of a thread, the modified interpreter on the local machine (the client) contacts one of the other machines also running the modified interpreter (the server). The client must then upload the class that is to be executed, along with any classes that may be referenced by this class, to the server. To do this, upon instantiation of a thread on a remote machine, replicas of all classes are made on the

remote machine. Then, consistency mechanisms ensure that the replicas on the different machines remain consistent during the course of the parallel computation.

Consistency enforcement occurs at the synchronization points in the program, using the notion of release consistency used in many DSM systems (Munin and Treadmarks, for example [6][1]). Release consistency assumes that all synchronization operations are classified as either acquire or release operations. The underlying coherence protocol relies on these acquire and release operations being visible to the system, and on the correct use of these acquire and release operations. The mechanisms used to provide consistency in ParaWeb make use of Java's built-in synchronization facilities. Each of the Java synchronization points is mapped onto the acquire and release operations of the release consistency model. That is, for the purpose of consistency, calling a synchronized method in Java is equivalent to an acquire operation, and exiting a synchronization method is equivalent to a release operation. Whenever a thread calls *wait* inside a synchronized method, that *wait* operation is treated like a release operation followed by an acquire. A *signal* operation is treated like a release operation.

The prototype implementation of ParaWeb is being designed with an update-based protocol, in which all replicas of a class are updated whenever a synchronization point equivalent to a release operation is invoked. Future enhancements to this protocol are planned which use more sophisticated coherence protocols.

4 An Example Application

As a test of our prototype implementation (using the parallel class libraries approach) we experimented with a simple parallel matrix multiplication program (blocking was not used). The program was executed on a network of 20 SUN SparcStation-10's connected by ethernet.

The execution time (in seconds) and speedup obtained using various numbers of machines to multiply two 400 by 400 matrices are shown in Table 1. The execution time using one machine was obtained by executing a sequential version of the program.

Machines	1	2	4	6	8	10	12	14	16	18	20
Execution Time	916	495	239	161	123	97	83	71	62	57	51
Speedup	1.0	1.9	3.8	5.7	7.4	9.4	11.0	12.9	14.8	16.1	18.0

Table 1: Matrix Multiplication Performance (execution time in seconds).

These results demonstrate the potential of this approach. Using a simple initial prototype we've been able to attain substantial decreases in execution times and good speedups. Additionally, the relative simplicity with which remote threads can be created and ease with which data can be communicated between machines (there are no concerns with respect to different architectures or the operating systems) offer significant incentives to continue pursuing this approach. However, simply examining speedup numbers can be quite misleading. This same 400 by 400 matrix multiplication application, written in C and compiled to native machine code executes on the same platform in 58 seconds. This is about 16 times faster than the sequential Java program can be interpreted on the same platform. Note that although Table 1 suggests that 18 machines would have to be used by the current ParaWeb system just to be able to execute as quickly as the optimized sequential C program, the development of high-performance interpreters and just-in-time compilers promise to improve this drastically.

We also believe that the incentives for some people to write Java programs will become strong enough that comparisons with the execution time of C or C++ programs will be of theoretical interest only. It is our goal to provide an environment for easily writing and efficiently executing high performance Java applications.

5 Issues

There are a number of issues that must to be dealt with before ParaWeb or other similar approaches will achieve wide-spread acceptance and use. However, many of these issues are not specific to ParaWeb but relate instead to the general issue of executable content over the Web. This includes, for instance, issues of security. On the surface, it would seem that there is an inherent danger in allowing clients to upload and execute programs on a server. However, the risk is no different than when applications are automatically downloaded and executed by clients browsing the Web. Thus, the inherent security features of Java should provide the same level of protection from malicious users. In addition, many computing sites may also likely restrict upload access to trusted users (for instance, users within the organization, or users that have previously registered with the site), thereby limiting the security risk.

Another issue, related to the performance of interpreted Java byte-code, is also beyond the scope of our own work. We expect that the introduction of just-in-time compilers will greatly reduce the performance gap for the types of compute intensive programs which are the target of ParaWeb. Short running programs may not necessarily benefit significantly from just-in-time compilation, but these programs are not likely to be either parallelized or executed remotely anyway.

Authentication and authorization are two other aspects of distributed and parallel computing that need to be further considered if resources spread throughout the Web are to be utilized in a safe and seamless fashion. We are not currently trying to develop solutions to these issues. The recent addition of pickling and remote method invocation (both are briefly described in Section 6) to version 1.1 of the JDK leads us to believe that the demand for authentication and authorization may warrant their addition to future JDK's.

6 Related work

The introduction of Java, with its architecture-independent and Web-centric programming framework, has sparked considerable interest among the distributed and parallel programming communities, although projects focusing on unifying Web technology and parallel computing are, like our own, still in their infancy. A recent proposal by Fox *et al* [3], for instance, proposes a system similar to ParaWeb called WebWork. They propose developing high-performance applications that make use of the Web's wealth of resources, by creating and utilizing compute servers throughout the Internet. While the implementation of WebWork and ParaWeb differ significantly (WebWork currently proposes using existing Web technologies such as CGI script), the motivation for both projects is derived from an interest in integrating the field of high performance computing with Web technology. The use of Java as a means for building distributed systems that execute throughout the Internet has also been recently proposed by Chandy it et al [2].

Some of the mechanisms that we are building into the Java interpreter in order to provide a distributed shared memory framework may eventually be realized through standard Java components. For instance, Riggs, Waldo and Wollrath [8], describe a *pickling* mechanism for Java that allows an object's state to be easily saved and restored. Although targeted at providing a means for marshaling and unmarshaling for remote object systems and object persistence, this pickling mechanism can be used for building a distributed shared memory system as well, since it would allow an object to be easily replicated and selectively updated when parts of the object are modified. Wollrath, Riggs, and Waldo, in a separate paper [10], also describe a Remote Method Invocation (RMI) mechanisms for Java based upon the pickling mechanisms. These pickling and RMI mechanisms will both be provided in an upcoming release of the JDK.

7 Summary

ParaWeb is a system for utilizing Internet or intra-net computing resources for executing serial programs on faster compute servers or parallel programs on a variety of possibly heterogeneous hosts. Our focus in ParaWeb is to provide efficient yet simple mechanisms for remote execution and communication within Internet applications. Many of the ideas we are exploring are extensions of existing load sharing, distributed

shared memory and message passing systems. Our early experiences suggest that ParaWeb can be implemented on top of conventional Web technology and may result in dramatic improvements in performance for coarse-grained applications.

8 Acknowledgments

We wish to thank the anonymous referees for their comments which helped us to improve the presentation of this work. We also wish to thank the Natural Sciences and Engineering Research Council (NSERC) for grants that partially supported this research.

References

[1] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *13th Symp. on Operating Systems Principles*, pages 152–164, October 1991.

[2] K.M. Chandy, B. Dimitrov, H. Le, J. Mandleson, M. Richardson, A. Rifkin, P.A.G. Sivilotti, W. Tanaka, and L. Weisman. A world-wide distributed system using Java and the internet. In *Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC5)*, Syracuse, NY, August 1996.

[3] G.C. Fox and W. Furmanski. Towards Web/Java based high performance distributed computing – an evolving virtual machine. In *Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC5)*, Syracuse, NY, August 1996.

[4] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley Developers Press, Sunsoft Java Series, 1996.

[6] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of USENIX*, 1994.

[7] T. Lindholm and F. Yellin. *The Java Virutal Machine Specification*. Addison Wesley Developers Press, Sunsoft Java Series, in preparation.

[8] R. Riggs, J. Waldo, and A. Wollrath. Pickling state in Java. In *2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 241–250, Toronto, Ontario, June 1996.

[9] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5), May 1990.

[10] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for Java. In *2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–231, Toronto, Ontario, June 1996.

[11] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogenous distributed computer systems. *Software: Practice And Experience*, 23(12):1305–1336, December 1993.