Javelin++: Scalability Issues in Global Computing

Michael O. Neary Sean P. Brydon Paul Kmiec Sami Rollins Peter Cappello

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{*neary, brydon, virus, srollins, cappello*}*@cs.ucsb.edu*

Abstract

Javelin is a Java-based infrastructure for global computing. This paper presents Javelin++, an extension of Javelin, intended to support a much larger set of computational hosts. First, Javelin++'s switch from Java applets to Java applications is explained. Then, two scheduling schemes are presented: probabilistic work stealing and deterministic work stealing. The distributed deterministic work stealing is integrated with a distributed deterministic eager scheduler. An additional fault tolerance mechanism is implemented for replacing hosts that have failed or retreated. A Javelin++ API is sketched, then illustrated on a raytracing application. Performance results for the two schedulers are reported, indicating that Javelin++, with its broker network, scales better than the original Javelin.

1 Introduction

Our goal is to harness the Internet's vast, growing, computational capacity for ultra-large, coarse-grained parallel applications. Some other research projects based on a similar vision include *CONDOR* [21, 13], *Legion* [18], and *GLOBUS* [14]. By providing a portable, secure programming system, Java holds the promise of harnessing this large heterogeneous computer network as a single, homogeneous, multi-user multiprocessor [6, 15, 1]. Some research projects that work to exploit this include *Charlotte* [5], *Atlas* [3], *Popcorn* [9], *Javelin* [12], and *Bayanihan* [23]. While there are many issues related to global computing, five fundamental issues that affect every Java-based global computing application are:

- *Performance* — If there is no niche where Java-based global computing outperforms existing multiprocessor systems, then there is no reason to use it.
- *Correctness* — If the system does not produce correct results, then there is no reason to use it.
- *Scalability* — In order for the system to outperform existing multiprocessor systems, it must harness a much larger set of processors. To do so, it must scale to a higher degree than existing multiprocessor systems, such as networks of workstations (NOW) [2].
- *Fault Tolerance* — It is unreasonable to assume that such a large set of components will have zero failures: Fault-tolerance must attend systems of this order.
- *Incentive* — Full use of global computing ultimately implies using a set of computers that is too large for any single person or organization to own or control. Where authority to command is lacking, incentives must be provided [10, 25]. To date, global computing has used fame, fun, or prizes as an incentive (e.g., the Great Internet Mersenne Prime Search [17], code-cracking (a money prize)[1], and SETI@home[2]). The Popcorn project [9] has explored computational markets.

Existing Java-based global computing projects have bottlenecks that currently prevent them from scaling to the thousands of computers that could be brought to bear. For example, the authors of Charlotte note:

> *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.*

Bayanihan [24] has limited scalability now. However, its authors note:

> *Currently, some ideas we are exploring include forming* server pools *to handle large numbers of clients, and using volunteer servers to form networks with more flexible topologies.*

Work apparently stopped on Atlas [3] after it had been tested using only a few workstations.

In this paper, we focus on scaling Javelin, comparing two scalable versions of Javelin, called Javelin++: one that schedules work deterministically, and another that schedules work probabilistically. Both versions work on a simple kind of adaptively parallel computation [11], called a *piecework* computation. Such an adaptively parallel computation decomposes into a set of sub-computations, each of which is communicationally autonomous, apart from scheduling work and communicating results. Piranha and Bayanihan, for example, are well suited to piecework computations. Raytracing is a well known piecework computation, often used by

[1] `http://www.rsa.com/rsalabs/97challenge`
[2] `http://setiathome.ssl.berkeley.edu`

global computing researchers. Matrix product also can be considered a piecework computation, since it can be decomposed into a set of block sub-products, whose results are simply added to produce the matrix product. Piecework computations are particularly attractive; they can get arbitrarily large, but their communication requirements are in harmony with global computing's intrinsic constraint: Internet communication latency is large.

The remainder of the paper is organized as follows: Section 2 briefly presents the Javelin architecture, and the architectural changes Javelin++ introduces. Section 3 discusses scalability in the context of global computing: dynamic code distribution, distributed task scheduling, including a distributed integrated work stealer/eager scheduler, and a basic fault tolerance scheme for replacing hosts that have failed or retreated. Section 4 presents the Javelin++ API, and illustrates its use on a raytracing application. Section 5 presents experimental results for the deterministic and probabilistic versions of Javelin++, indicating the sensitivity of their scalability to granularity. The final section concludes the paper, indicating some immediately fruitful areas of global computing research and development.

2 Architecture

The Javelin++ system architecture is essentially the same as its predecessor, Javelin [12]. There are still three system entities — clients, brokers, and hosts. A *client* is a process seeking computing resources; a *host* is a process offering computing resources; a *broker* is a process that coordinates the allocation of computing resources. We did, however, introduce a few changes to the architecture. The most important ones are:

- Communication is now based on Java RMI instead of TCP sockets. The application programmer thus no longer needs to implement a communication protocol[3]. Of course, the use of RMI requires the presence of JDK 1.1.x or later or compatible software at any host participating in Javelin++.

- For a number of reasons, we found it desirable to base our system on Java applications instead of applets, as was done before. This is probably the most prominent architectural change in the new system. The reasons that compelled us to make this switch are outlined below.

- Javelin++ is the first version that actually implements a distributed broker network. Although the concept was already included in the old architecture, it was never practically achieved. Section 3.1 talks about the broker network.

In the remainder of this section, we first briefly recap the architecture of the old Javelin system. This is followed by a discussion of the advantages and disadvantages of using Java applications instead of applets.

2.1 The Javelin Architecture

Figure 1 illustrates our architecture. Clients register their tasks to be run with their local broker; hosts register their intention to run tasks with the broker. The broker assigns tasks to hosts that, then, run the tasks and send results back to the clients. The role of a host or a client is not fixed. A machine may serve as a Javelin host when it is idle (e.g., during night hours), while being a client when its owner wants additional computing resources.

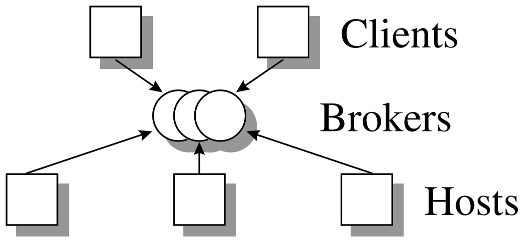

Figure 1: The Javelin Architecture.

One of the most important goals of Javelin is *simplicity*, i.e., to enable everyone connected to the Internet or an intranet to *easily* participate in Javelin. To this end, the design is based on widely used components: Web browsers and the portable language Java. By simply pointing their browser to a known URL of a broker, users automatically make their resources available to host parts of parallel computations. This is achieved by downloading and executing an applet that spawns a small daemon thread that waits and "listens" for tasks from the broker. The simplicity of this approach makes it easy for a host to participate — all that is needed is a Java-capable Web browser and the URL of the broker.

Tasks are represented as applets embedded in HTML pages. This design decision implies certain limitations due to Java applet security: E.g., all communication must be routed through the broker and every file access involves network communication. Therefore, in general, coarse-grained applications with a high computation to communication ratio are well suited to Javelin. For more information on the original Javelin prototype, see [12].

2.2 Java Applets vs Applications

As the underlying distributed object technology, Java RMI (Remote Method Invocation) is used. In the original Javelin prototype, all applications run as Java applets, which has the advantage of extreme ease of use from the point of view of a participating host — the user only needs to point a Java-capable browser to a broker's web page to get started. Another advantage of using applets is the strict security model: the applet is effectively sandboxed by the browser. A user can trust the established security policy of his or her favorite browser when running untrusted code. However, the use of applets in this context has some serious drawbacks:

- *No local file I/O* — applet security generally does not permit access to the local hard drive, making it hard if not impossible for some applications to run. For instance, in the case of seismic data processing, local file I/O is necessary to read large amounts of data in parallel.

- *No direct point-to-point communication* — since applets may not accept any connections, and may only initiate connections back to the server from where they were downloaded, point-to-point communication can

[3] JavaParty [22] and HORB [19] are alternatives to RMI, which however lack RMI's widespread installed base.

be achieved only by providing special routing servers that relay all messages sent by Javelin hosts. This is a serious performance drawback, and the task of implementing the network routing service is tedious, especially with respect to fault tolerance.

- *No native code interface* — applets are not allowed to interface with native code on the host machine, and uploading native libraries through the browser is not possible. That means that any kind of commercial off-the-shelf (COTS) application cannot be made to run on the infrastructure, although there are such applications that would benefit greatly from global computing (e.g., the *Bryce* image rendering software was recently implemented as a standalone global computing application including a simple trading space [8]). There is also an immense body of legacy code in scientific applications written in Fortran or C[4] which could benefit from our infrastructure.
- *No standard platform* — increasing browser heterogeneity makes it hard to program to all platforms, defeating the original idea of Java's platform independence. Since the arrival of JDK 1.1, browser developers have been sluggish in implementing the complete API, leading to various subsets of the JDK being supported by different platforms. A prominent example is Microsoft's outright denial of support for Java RMI in Internet Explorer, making it impossible to use the most convenient and natural object technology for Java in conjunction with their browser. As of today, JDK 1.0.2 remains the only agreed upon standard.

We consider these disadvantages so severe that they threaten the general usefulness of Javelin++, since they disallow the implementation of many interesting applications on our platform. Therefore we decided to switch to Java applications running on Sun's JDK 1.1 (or later) as the main platform for Javelin++ applications. This switch enables us to overcome all the above disadvantages. However, such a switch has its own disadvantages.

First, the user must have JDK 1.1 installed on any machine that is to become a Javelin++ host. Since JDK is widely distributed at present, we do not consider this a serious drawback. The main disadvantage of applications is, of course, the lack of a security model that is predefined, agreed upon, and therefore comfortable for the user, raising the question of trust in the Javelin++ system. This can be overcome in two ways:

1. A Javelin++ security model can be provided by implementing a *SecurityManager* class that ensures that applications communicate only with other Javelin++ applications, and have limited access privileges to local resources like the file system.
2. On certain operating systems, e.g., Solaris and Linux, it is possible to sandbox a process externally through the so-called "/proc" interface. This has been successfully demonstrated in the Berkeley Janus project [16] and the UCSB Consh project [20].

Both approaches can lead to an even more secure execution environment than the browser itself can provide. For instance, the experiment of the Knitting Factory project [4] found that when using Java RMI at least one browser, Sun's HotJava, permits direct point-to-point communication between applets once RMI handles have been exchanged through the server!

Compared to applets, applications might appear to lack ease of use for hosting users. However, we are pursuing the screen saver approach that arguably is even easier for hosting: Users would download and install a Javelin++ screensaver on the host machine. This screensaver runs the JVM and the Javelin++ daemon while the host machine is idle. In this way, the user need not do anything to invoke the host; it is done for him precisely when his machine is available. A Unix daemon functioning similarly also is being developed. Although this has the disadvantages of requiring user installation, and developer implementation for each OS, thus losing some of the platform independence of Java, it makes hosting essentially effortless.

3 Javelin++: A Scalable Architecture

In this section we present our approach of a scalable global computing system. Other projects have tried or are currently trying to achieve greater scalability, e.g., Atlas [3] through its tree-based approach, and Bayanihan [24] with its volunteer server concept; but to date, no large-scale experiments have shown that these concepts work in practice. The original Javelin achieved good results up to about 60 hosts, when the single broker/router bottleneck became noticeable.

Without modifying the original Javelin architecture, Javelin++ introduces a number of scalability enhancements, described below. The most prominent are:

- a distributed broker network that overcomes the single broker bottleneck and permits much greater host participation,
- the switch from Java applets to applications as described in Section 2, which permits point-to-point communication and thus allows arbitrary graph configurations, and
- two different schemes of work distribution, a probabilistic one and a deterministic one, that both offer the potential to accommodate large numbers of hosts participating in a single application.

Let us begin by clarifying what we mean by *scalable*: If a global computational infrastructure is scalable, its components have bounded power — bounded computational rate, bounded communication rate, and bounded state[5]. In particular, for Javelin++ to be scalable, its clients, brokers, and hosts have bounded power. These bounds imply that, for example, clients, brokers, and hosts, can communicate with only a fixed number of other components during a fixed interval of time. Thus, at any point in time, there are bounds on the number of connections between hosts, between brokers, between brokers and hosts, and between the client and brokers. Bounded state similarly implies bounds on the number of brokers that a broker can know about at any point in time.

The Javelin prototype offers just a single broker/router that becomes a bottleneck when too many hosts participate in a computation. Clearly, a network of brokers must be created in order to achieve scalability. Internet-wide participation means that all hosts must be largely *autonomous* and

[4] This type of code could be called "SOTS" — scientific off-the-shelf.

[5] In this context, bounded stands for *bounded by some constant.*

able to work in the presence of node and network failures. Scalability implies that the architecture cannot be centralized. Bounded state implies that no site can, in general, have a global system view (e.g., a table with the names of all participating brokers). We have identified two key problems in building a scalable architecture:

1. Host allocation and code distribution — How does a client find hosts for its computation, and how does the code get distributed efficiently to a potentially very large number of hosts?

2. Data communication at runtime — How is data exchanged between participating hosts after an application has been successfully started?

In the following we describe our approach to solve these problems. The section is structured according to the different states a Javelin++ host can be in during its lifetime. The complete state transition diagram is shown in Figure 2. There are four states: NoHost, Standby, Ready, and Running. If a host has not joined Javelin++ it is in state NoHost. The transition to Standby is made by downloading and starting the Javelin++ daemon and then registering with a broker. In the next section we describe how brokers are managed, hosts are allocated, and code is shipped so that an application is ready to start, causing a state transition from Standby to Ready. In Section 3.2.2 we present two different data exchange mechanisms that allow the host to run the application and therefore transition to Running. The first is a probabilistic approach based on a *distributed, double ended queue* and address hashing; the second is a deterministic, tree-based approach. The performance of these two approaches is compared in Section 5.

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

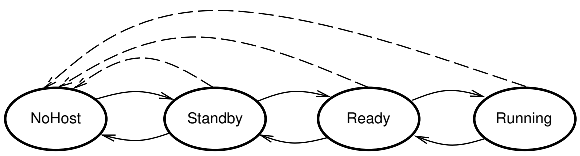

Figure 2: State Transition Diagram for Javelin++ Hosts.

3.1 Scalable Code Distribution via a Broker Network

3.1.1 Network Topology and Broker Management

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

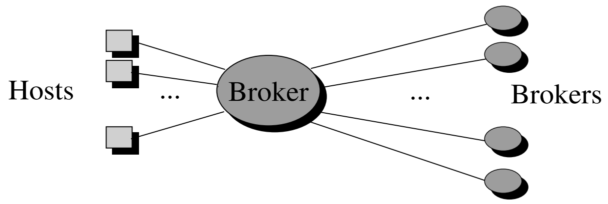

Figure 3: Broker Connections.

In principal, a broker is just another Javelin++ application. That means that it runs on top of the Javelin++ daemon thread. However, since brokers are expected to be a lot more stable and reliable than other hosts, certain conditions have to be met: A broker must run on a host with a "permanent" connection to the Internet, i.e., slow modem connections are not acceptable, and the user donating a broker host must be prepared to run the broker for a "long" duration and give the system "ample warning" before withdrawing the host, so that connected hosts can be moved to other brokers.

We distinguish between two types of broker: *primary brokers* and *secondary brokers*. Technically, there is not much difference, except for the way the broker starts up. A primary broker is a broker that starts up without logging in to another broker as a host first. This is to guarantee that there is a minimal broker network at system startup. Primary brokers can start up from shell commands and link to other primary brokers by reading in a configuration file. In contrast, secondary brokers start up as normal Javelin++ hosts by linking to their local broker. At registration time the host indicates whether or not it is prepared to run a broker according to the above rules.

A secondary broker comes to life when the broker it is connected to exceeds its individual limit for host connections. In order to accommodate the host that causes this overflow, the broker chooses one of its hosts that is prepared to be a broker and preempts the application running on that host. Then it sends the broker code to the new broker host and moves some of its hosts to the new broker. Also, the new broker gets connected to other brokers by using the same (or part of the same) configuration file of the primary broker which is also sent to it by the old broker. All this can be achieved through the Javelin++ daemon. Next, the daemons of the hosts that were moved are notified of their new broker. This should be entirely transparent to the users who donated the hosts. In the same way, the system can collapse again if the number of hosts connected to the secondary broker drops below a certain threshold, say e.g. 25% of its host capacity.

3.1.2 Code Distribution

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

If every host that participates in a computation had to go to the client to download the code this would soon lead to a bottleneck for large numbers of hosts. Therefore, first the local broker and then every other broker that joins in a computation will act as a *cache* on behalf of the client.

The loading and caching mechanism is implemented as a modification to the standard Java *ClassLoader* — whenever a *loadClass()* command fails at a host it is translated to an RMI call to the local broker, which in turn will either deliver the requested class from its cache or make a recursive RMI call to the broker it retrieved the application from. If all calls in this chain fail to deliver the requested class, the client will finally be contacted and deliver the original class file, which will then be cached at all intermediate brokers in the chain. Subsequent requests by other hosts will not reach the client again, thus eliminating another bottleneck in the system.

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

1. The client registers with its local broker.

2. If the broker is willing to accept jobs, the client then sends a description of the application to the broker[6]. Depending on the type of application, the client may now start up and execute on its own.

3. A host joins the system by downloading the Javelin++ daemon class and starting a JVM that executes the daemon.

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

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

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

3.2 Scalable Computation

After distributing the code successfully, we can now tackle the next problem of managing a scalable computation. In Javelin++ we follow two distinct approaches to solve this problem, a probabilistic and a deterministic model. Whereas the probabilistic approach is somewhat "chaotic" in the sense that communication between hosts is completely unstructured, the deterministic approach structures the participating hosts into a tree in which some hosts become "managers" for other hosts. Both approaches offer high potential for scalability, and a performance comparison is attempted in Section 5. We now give a brief description of our strategies.

3.2.1 Work Stealing

The fundamental concept underlying both our approaches is *work stealing*, a distributed scheduling scheme made popular by the Cilk project [7]. Work stealing is entirely demand driven — when a host runs out of work it sends a work request to some other host it knows. How exactly that host is selected depends on whether the probabilistic or deterministic strategy is used, see below. One advantage of work stealing is its natural way of balancing the computational load, as long as the number of tasks remains high in relation to the number of hosts; a property that makes it well suited for adaptively parallel systems.

[6]currently consisting of the name of the application class and the ID of the client

For our work stealing schedulers we use two main data structures that are local to each host: a *table of host addresses* (technically, Java RMI handles), and a *distributed, double-ended task queue* containing "chunks of work". For the reader who knows our previous Javelin prototype [12] the deque will sound familiar. Indeed we have only further refined this approach since it promised good scalability from the beginning.

The working principle of the deque is as follows: the local host picks work off one end of the queue, whereas remote requests get served at the other end. Jobs get split until a certain minimum granularity — determined by the application — is reached, then they are processed. When a host runs out of local jobs, it picks one of its neighbors from its address table and issues a work request to that host.

3.2.2 The Probabilistic Approach

In the probabilistic model, the host selects the target of its work request at random from the list of hosts it currently knows. In doing so the host piggybacks its own address information onto the request so that address information can propagate through the set of participants. Regardless of whether the request is successful, the callee returns a constant number of his own addresses for the same purpose. The caller will then merge its address table with the set of returned addresses. Thus, its knowledge of participants will increase until its table fills up and "older" addresses must be evicted, which can be taken care of by a standard replacement policy like, e.g., LRU. All this will result in a *working set* of connections for each host.

Since the list of known hosts can grow relatively large for big applications, a *hash table* is used to store the addresses. From the point of view of scalability, using a hash table allows for fast retrieval in the average case and scales to very large numbers. In addition, there is no centralized site in this setup, and host autonomy is guaranteed since sufficient information is kept locally to remain functional in the presence of failures. It is important to observe that the address table is *bounded in size* — the hash table is preallocated to some fixed size that remains manageable.

3.2.3 The Deterministic Approach

Our second approach implements a deterministic scheme. Here, we use a balanced tree — similar to a heap — as the underlying structure of our deterministic model. Again, the fundamental concept employed is work stealing. The main difference is that the selection of the host to steal work from follows a deterministic algorithm based on the tree structure.

Initially, each host retrieves a piece of work from its parent and will perform computation on the work one piece at a time. When a host is done with all the work in its deque, it will attempt to steal work, first from its children and, if that fails, from its parent. We chose this strategy to ensure that all the work assigned to a subtree gets done before a node requests new work from its parent. To facilitate this scheme, each host keeps a counter of the total work assigned to its subtree, plus a counter for each of its children. It is important to observe that the counters are not likely to reflect the exact state of the tree, but rather serve as *upper bounds* on the amount of work left in a subtree. This way, a host can make an "educated guess" as to which of its children is most likely to have more work, and direct its request to that child first. The counters are updated on each reply to a work request.

Work stealing within a balanced tree of hosts ensures that each host gets amount of work commensurate with its capabilities. The client is always located at the root of the tree and manages the tree, i.e., it maintains a heap data structure for that purpose. When a new host joins, it is assigned a leaf position in the tree by the tree manager. The tree fanout can be chosen individually for each application at startup.

Figures 4 through 6 illustrate the deterministic work stealing process. In Figure 4, three hosts have joined in the computation of a simple raytracing scene. At first, only Host 0 — the client — started work on the whole image. Then, Host 1 joined and stole the left half of the image from Host 0. Next, Host 2 joined and stole the upper right quarter of the image from the client. The figure depicts the situation when all hosts have already completed some atomic pieces of their work. Consequently, the work counters at Host 1 and Host 2 show the exact number of atomic pieces remaining in the deques of these hosts, 6 and 3, respectively.

The client only has an upper bound on these counters, indicating the initial amount of work given to each of its children — 8 and 4, respectively. However, it does know that the client itself has completed 3 units of work, so it can adjust the upper bound on the total work — its own work counter — to a value of 13.

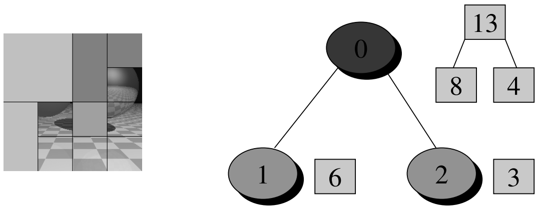

Figure 4: Work Stealing: 3 hosts working on a raytracing scene.

In Figure 5, the computation has progressed to the point where first the client was able to steal the upper left quarter of the image back from Host 1, and then Host 1 in turn stole the upper left eighth of the image back from the client. Host 2 still has work remaining from its initial share of the image. Through this exchange the client has gained information on the amount of work Host 1 has completed, and therefore it knows that there can be at most 7 pieces left to compute, with at most 2 pieces for Host 1 and, since it has not heard from Host 2 yet, at most 4 pieces for that host. Of course, Hosts 1 and 2 know the exact workloads in their deques, just as before.

Finally, Figure 6 shows the end of the computation with all counters down to 0.

3.3 Fault Tolerance

For fault tolerance purposes, the current version of Javelin++ employs *distributed eager scheduling*, where pieces of work can be reassigned to idle hosts in case results are still outstanding. Eager scheduling was made popular by the

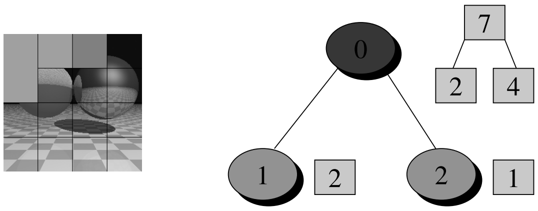

Figure 5: Work Stealing: Client and Host 1 have stolen work from each other.

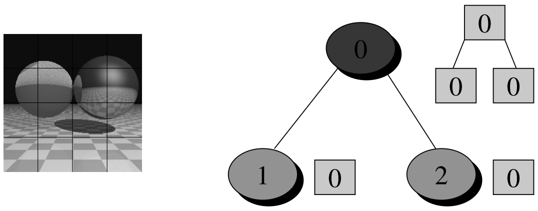

Figure 6: Work Stealing: Image complete.

Charlotte project [5] and has also been used successfully in Bayanihan [24]. It is a low overhead way of ensuring progress towards the overall solution in the presence of failures or varying processor speeds.

We first explain our distributed eager scheduling strategy, then we give a brief description of an additional fault tolerance feature of Javelin++ — detecting and correcting host failures in the deterministic tree.

3.3.1 Distributed Eager Scheduling

The Javelin++ eager scheduling logic is located on the client. Although this may seem like a bottleneck with respect to scalability, it is not, as we shall explain below. The mechanism employed is lightweight, introducing little overhead, as our experiments in Section 5 demonstrate.

The basic data structure required is a heap-like *problem tree*, which the client maintains to keep track of the computation status. The tree has a node for every piece of work the problem can possibly generate, and is structured as follows: at its root is the complete, undivided problem itself; its children are the subproblems resulting from a single split of the root problem; and so on, until the atomic pieces of the problem appear at the leaves of the tree. Each node can be in one of three states: *done*, meaning the result for the subproblem has been received by the client; *partly done*, meaning that results have been received by the client for some but not all descendants of this subproblem (i.e., some but not all subproblems of this subproblem); and *undone*, meaning that no results whatsoever have been received by the client for this subproblem. Initially, all nodes are in the *undone* state. The processing itself consists of two distinct routines:

1. *Result Processing* — this is the "normal" mode of operation, in which the client records all incoming results. Specifically, the client marks the subproblem corresponding to the incoming result as *done*, and then recurses up the tree marking ancestors of the subproblem either *done* or *partly done* depending on their current status.

2. *Eager Scheduling Selection* — this routine is invoked only when regular work stealing fails for the client, i.e., the *client* cannot get work from its children in the host tree. In this case, the client selects the next piece of work marked *undone* for rescheduling. Since the tree is organized as a circular array with piece sizes monotonically decreasing, this piece is guaranteed to be the largest available undone piece. The next time selection is invoked, it will proceed from the current point to select the next largest undone piece, and so on. At the end of the array, the selection process simply wraps around to the beginning and enters another round.

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

Figures 7 through 9 give an example of Javelin++ eager scheduling. In Figure 7 we see how the result of the atomic piece with ID 4 arrives at the client. The client subsequently marks all its ancestors including the root as *partly done*. In

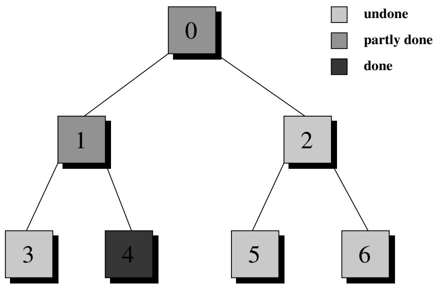

Figure 7: Eager Scheduling: Result 4 arrives.

Figure 8 another result arrives for the atomic piece with ID 3. Now the client can also mark the parent of this piece, node 1, as *done*. Finally, in Figure 9 we show how, assuming no further results arrive at the client and work stealing has failed, the client selects piece number 2 as the largest *undone* piece of work. This piece will now be reissued for host work. It subsequently may be split, and parts of it stolen by other hosts.

Our variation of eager scheduling principally works with both scheduling approaches. In the deterministic case, eager scheduling proceeds in rounds. Work is reissued, starting a new round, only when all work in the current round has been split into atomic pieces and all atomic pieces have been assigned to some host. In the probabilistic case, however, it is difficult to detect when all work in the current round has been split into atomic pieces and all atomic pieces have

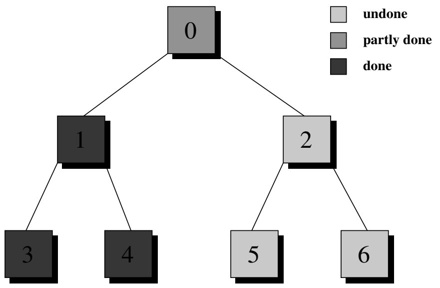

Figure 8: Eager Scheduling: Result 3 arrives.

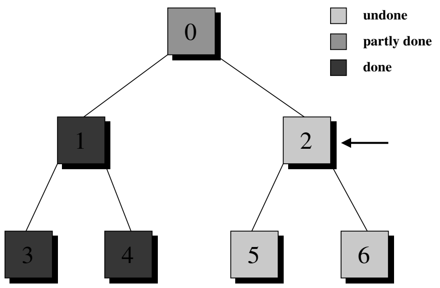

Figure 9: Eager Scheduling: Piece 2 selected.

been assigned to some host. Rather, work is reissued in the probabilistic case when the client is out of work and fails to find work after k attempts to steal work from randomly selected hosts (for some k, a parameter of the probabilistic eager scheduling strategy).

3.3.2 Fixing the Deterministic Tree

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

Javelin++ provides a feature that automatically fixes a tree as soon as a host failure is detected. As a precondition we make the assumption that the client is a *stable participant*. This is reasonable because the client is the initiator of the whole computation.

The tree repair scheme works as follows: When a host is assigned a position in the tree, it is given a list of all of its ancestors, with the client being the first element of the list. If a host detects that its parent is dead, it immediately notifies the client of this condition. If the empty position has already been reported and filled, the tree manager traverses the tree representation, and returns a new ancestor list to

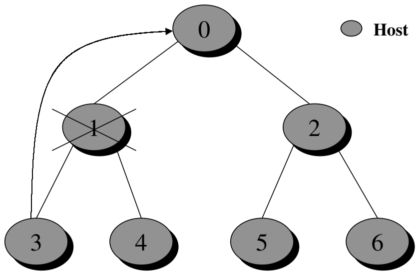

Figure 10: Host 1 fails, Host 3 detects failure.

the host. However, if the host is the first to report the failure, the tree manager reheaps the tree. First, it notifies the last node in the tree, which is to be moved to fix the gap. Figures 10 through 12 illustrate the situation where node 1 has failed and is replaced by node 6.

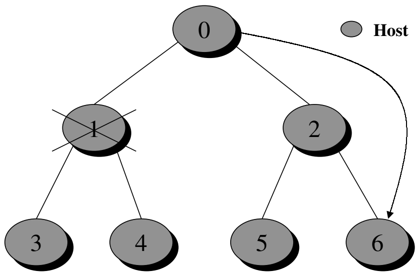

Figure 11: Client preempts Host 6.

Node 6 is assigned a new list of ancestors, and is moved to its new position. Then, the tree manager traverses the tree representation to find the new ancestor chain of the orphaned node, and returns that chain.

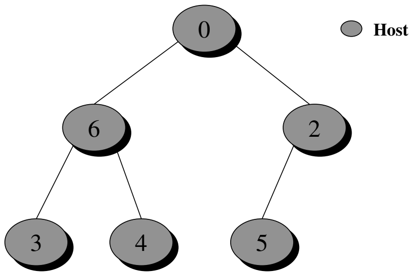

Figure 12: Host 6 takes over for Host 1.

Currently, the tree is managed by a single entity and therefore presents a potential bottleneck if the host failure rate is high. However, it would be possible to modify the existing implementation to distribute tree management throughout the broker network. In this case, the host failure rate which the system could recover from would increase as the number of brokers increased.

At present, the tree repair scheme can only cope with host failures, i.e., all hosts that detect a failure must agree on that observation. A single host is not able to distinguish between host and link (communication) failures; the result is the same from the point of view of the host. In case of a link failure between a host and only one of its children, the present scheme reports a failure to the client even when sibling hosts can still communicate with the "failed" parent host. Clearly, what is needed is a form of *quorum consensus* algorithm. Therefore, our basic scheme needs to respond in a more sophisticated way to link failures. This is a current topic of research in Javelin++.

4 The Javelin++ API

In this section we illustrate our system from an application programmer's point of view. We first present the classes needed by the programmer to create a Javelin++ application. We then give an example that shows how the API is actually used and how easy it is to create Javelin++ applications.

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

All of the Javelin++ classes are contained in two packages: **JavelinPlus** and **JavelinPlus.util**. The first package contains all of the core Javelin++ classes and the second one contains data managers and other helper classes. We follow the convention that all classes and interfaces beginning with the letter "J" are implemented in Javelin++ and can be directly used by the application, whereas interfaces not beginning with "J" must be implemented by the application in order to work with the system.

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

4.1 The JavelinPlus Package

This package contains all the core classes needed by clients, hosts, and brokers, including the Javelin++ daemon mentioned in Section 3. The programmer writing an application for Javelin++ only needs to get acquainted with the `JavelinClient` class.

```
public class JavelinClient {
  public JavelinClient(String client,
                       String className,
                       String broker);
  public void begin();
  public void terminate();
}
```

Javelin++ clients must instantiate the `JavelinClient` class. The only constructor of `JavelinClient` takes the local host name, the top-level class name used to load the host classes, and a broker's host name. Once the client is ready to start the computation, the client invokes the `begin()` method. This causes the client to register with a broker, which in turn allows for the broker network to assign hosts to the client's computation. The `terminate()` method unregisters the client allowing the broker network to clean up and stop assigning hosts to that client. It is typically called after the computation is done and before the client exits.

4.2 The JavelinPlus.util Package

To manage the computation, clients and hosts must instantiate one of the data managers in this package. Data managers dictate how the computation is divided, how hosts obtain work, and how results return to the client. As discussed in Section 3, data managers can either be probabilistic or deterministic, and they are responsible for providing scalability and fault tolerance. Currently, Javelin++ provides two data managers: the deterministic `JavelinDDeque` and the probabilistic `JavelinRDeque`. Both of these implement the `JDataManager` interface shown below.

```
public interface JDataManager {
  public void addWork(Splittable work);
  public Splittable getWork();
  public void returnResult(Object result);
  public void setResultListener(ResultListener rl);
  public void setDoneListener(DoneListener dl);
}
```

The three main methods are `addWork()`, `getWork()` and `returnResult()`. In our model, a host uses the first method to pass new work to the data manager. In the piecework scenario this method is typically only executed once by the client to initialize the computation. The `getWork()` method is used by a host to obtain a piece of the computation. In case the computation produces a usable result, the host passes that result to the client using the `returnResult()` method. However, the exact way that the result propagates to the client depends on the underlying data manager. For instance, atomic results can be sent directly to the client or they can be collected, composed, and sent as composite results.

The programmer must also tell the data manager how to notify his application whenever a new result arrives and when all the work is complete. This is done by the methods `setResultListener()` and `setDoneListener()`. The two methods are mainly needed on the client which needs to process results and is interested in knowing when the computation is complete. For this purpose, the programmer must implement the two interfaces below so that the respective methods can be called by the system.

```
public interface ResultListener {
  public void handleResult(Object result);
}

public interface DoneListener {
  public void workDone();
}
```

We now mention how the client conveys its work to a data manager. For this, the programmer defines a class, representing the type of work to be done, that implements the `Splittable` interface, shown below. The data manager uses the Splittable methods to divide and distribute the work to hosts.

```
public interface Splittable {
  public boolean canSplit();
  public Splittable[] split();
  public int getObjectSize();
}
```

The `split()` method should split the work represented by a particular object into two relatively equal parts. The two parts are returned in an array of length two[7]. For example, assume we have a class that implements the `Splittable` interface and represents an image. If we were to invoke the `split()` method on an instance representing an n by n image, the returned array should contain two new instances each representing an $\frac{n}{2}$ by n image. The `canSplit()` method determines if a split is possible and is always invoked prior to `split()` method. If `canSplit()` returns *false*, the `split()` method will not be called. Finally, the `getObjectSize()` method returns the integer size of the piece of work: the number of atomic pieces into which this piece decomposes. This is needed by the deterministic deque which keeps integer counters of all work assigned to a tree node and its children. The method is ignored by the random deque.

4.3 Examples

The main design goal is to separate the computation engine from the data delivery. The data delivery interacts with Javelin++ to obtain and format the work for the computation engine. This design produces two very desirable properties. First, we can reduce application writing to using an off-the-shelf program/library (computation engine) and only writing a small data delivery part. Second, having done one such application, it is very easy to change to a different computation engine.

The client must pass the name of the host class into the `JavelinClient` constructor. This class has to implement the `Runnable` interface, since the Javelin++ daemon is going to execute the host application as a thread. Therefore, the programmer must implement the `run()` method, which is the first method that is going to be invoked.

Prior to the computation, the host is only required to instantiate the same data manager as the client. Then, the host starts the computational loop: ask data manager for work, compute work, and return results. Once the data manager returns *null*, indicating that there is no more work, the host can terminate by simply returning from the `run()` method.

The skeletons for the client and the host are presented below. To save space and increase readability much of the error handling code has been omitted.

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

  public GenericClient(String broker) {
    jClient = new JavelinClient(localHost,
                                "GenericHost",
                                broker);
    // Create a work object of the class
    // that implements Splittable.
    work = new ...;

    // Create a data manager.
    // Here, a deterministic deque
    // is instantiated.
    dm = new JavelinDDeque();
```

[7] although other ways of splitting are conceivable with this interface!

```
    // Pass the work to the data manager.
    dm.addWork(work);
    dm.setResultListener(this);
    dm.setDoneListener(this);

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

  public void handleResult(Object result) {
    ...    // ResultListener Implementation.
  }

  public void workDone() {
    // DoneListener Implementation.
    jClient.terminate();
  }

  public static void main(String[] argv) {
    GenericClient genClient
                 = new GenericClient(argv[0]);
  }
}

public class GenericHost implements Runnable {
  JDataManager dm = null;

  public GenericHost() { ... }

  public void init() {
    // Instantiate the same data manager
    // as in the client.
  }

  public void run() {
    init();

    // Computational loop.
    while ((Object work = dm.getWork()) != null) {
      Object result = doWork(work);
      dm.returnResult(result);
    }
  }
}
```

Next, we give some code extracts from our raytracing application. The raytracer is still the same application that was used in the original Javelin system [12]. We first show how this application implements the `Splittable` interface to tell Javelin++ how objects can be split. Here, the `RectEntry` class shown below simply extends the `java.awt.Rectangle` class to define the area that needs to be rendered.

```
public class RectEntry extends java.awt.Rectangle
  implements Splittable {
  // minimum size for split
  public static final int limit = 32;
  private int jsize = 0;

  public RectEntry(int x, int y, int ww, int hh) {
    super(x, y, ww, hh);
    jsize = (ww/limit) * (hh/limit);
  }

  boolean canSplit() {
    return (width > limit || height > limit);
  }

  Splittable[] split() {
    Splittable[] result = new Splittable[2];

    if (width > height) {
      result[0] = new RectEntry(x, y,
                                width/2, height);
      result[1] = new RectEntry(x + width/2, y,
                                width/2, height);
    }
    else {
      result[0] = new RectEntry(x, y,
                                width, height/2);
      result[1] = new RectEntry(x, y + height/2,
                                width, height/2);
    }
    return result;
  }

  public int getObjectSize(){
    return jsize;
  }
}
```

Finally, the computational loop for the raytracer is shown below. First, we ask the data manager for an area to render. Our rendering engine is simple and is totally contained in the `RayTrace` class. To render an area, the data delivery simply invokes the `raytrace()` method. At the end of the loop the result for this area is returned.

```
while ((area = (RectEntry)dm.getWork()) != null) {
  int [] pixels
       = tracer.raytrace(area.x, area.y,
                         area.width, area.height);

  dm.returnResult(new RectResult(area, pixels));
}
```

5 Experimental Results

Tests were run in campus student computer labs under a typical workload for the network and computers. This environment consists of

- 12 Pentium II, 350 MHz processors with 128 MB RAM,
- 41 Pentium-S, 166 MHz processors with 64 MB RAM, and
- 10 Sun UltraSparc with 64 MB RAM and processor speeds varying from 143 to 400 MHz.

All machines run under Solaris 2.5.1 and 2.6. The machines are connected by a 100 Mbit network. We used JDK 1.2 with active JIT for our experiments, although our code is designed to run with JDK 1.1.x as well.

To test the performance of the probabilistic and the deterministic deque, we ran experiments on the raytracing application described in Section 4. The performance was measured by recording the time to render the image. The size of the image used for testing is 1024 x 1024 pixels, consisting of a plane and 993 spheres arranged as a cone. This image is complex enough to justify parallel computing, but small enough to enable us to run tests in a reasonable amount of time. The image is recursively decomposed by the splitting process described in Section 3 into a set of atomic subimages. The size of such an atomic piece can be chosen freely by the application, e.g., 128 x 128 pixels. Each sub-image constitutes an independent task for a host to compute. The computational complexity of a task thus depends on the size of the sub-image and its complexity (i.e., the number of objects in the scene to be raytraced).

The test image took approximately 6 hours to render on one machine. Since the image being tested was an actual scene, the computational complexity of an individual task varies, depending on which part of the scene it represents. We manually joined hosts to the computation soon after the client started. In the future, we plan to have hosts in place, waiting for a client to start.

In the first experiment, we varied the number of hosts on the image decomposed into 1024 sub-images of 32 x 32 pixels. In our second experiment, we fixed the total work (image) and number of hosts, and varied the task size, thus varying the number of tasks to be distributed. In a production environment, we would set the deterministic tree's branching factor, or fanout, to maximize efficiency. For test purposes, the tree's branching factor was set to 4, much less than its maximum efficiency, to force the tree to have some depth. Our third experiment compares the performance of the deterministic deque with eager scheduling against its performance without eager scheduling, thus measuring the overhead of the result processing in the absence of failures.

5.1 Measurements

Before we discuss our results, it is necessary to clarify what we mean by the term "speedup" in our setting. Traditionally, speedup is measured on a dedicated parallel multiprocessor, where all processors are homogeneous both in hardware and in software configuration, and varying workloads between processors do not exist. Obviously, in such a setting speedup is well defined as T_1/T_p, where T_1 is the time a program takes on one processor and T_p is the time the same program takes on p processors.

Therefore, strictly speaking, in a heterogeneous environment like ours the term speedup cannot be used anymore. Even though we tried to run our tests in as much a homogeneous setup as possible — we ran all experiments involving up to 32 hosts only on our slowest but most numerous machines, the Pentium 166 processors — the varying workloads on both the OS and the network amounted to big differences in the individual computational powers of hosts.

However, from a practical point of view, a user running a client application on Javelin++ that is joined by a large set of hosts will definitely see "speedup"; the application will simply run faster than on a single machine. This speedup can very well be quantified in the same way as before, only that it is now relative to the specific client machine, i.e., a user sitting at a fast machine will experience smaller speedup than someone using a less powerful computer. Also, this type of speedup is not a constant, since on another run, even with the same set of hosts, the running time is bound to be different. From our experiments we can say that the running times we measured were relatively consistent for every given configuration — an experiment involving 16 hosts, for instance, would yield almost the same performance if repeated with the same parameters, with results varying by at most 20%.

Having considered all this, we believe that we can still speak of speedup in this context. We would like to suggest the term *practical speedup* in this new setting as a means of distinguishing between the two scenarios. In the following, we may omit the word "practical" when the meaning is clear from the context.

In Figure 13, our base machine is a slow Pentium 166. In this case, a user can experience considerable speedup. Our results show that, for both the deterministic and the probabilistic deque, the speedup degrades only slightly from linear for the higher host numbers.

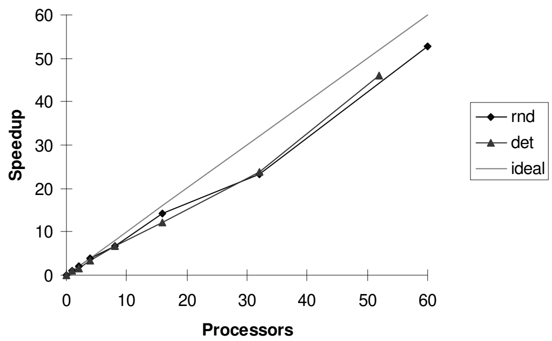

Figure 13: Practical Speedup for Random and Deterministic Deques compared against a slow client.

The slight degradation, we believe, is associated with transient phenomena: starting up the large number of processors and winding down the computation, where necessarily a large number of processors eventually must run out of work. Also, the varying workloads of participating hosts reduce the potential speedup, as results from slow processors come in late. Work stealing can alleviate this problem, but overall a loss in performance cannot be overcome.

One phenomenon has to be explained in this context: both curves first show a degradation up to 32 hosts, then a sudden increase in speedup for the largest experiments. This is because only for our largest experiments (more than 41 processors), we ran out of slower processors and added the fewer but faster processors, which managed to steal more work from slower processors and thus improved speedup. For the random deque, our best result was a speedup of 52 for 60 processors. The deterministic deque achieved a comparable speedup of 46 for 52 processors. Thus, our comparison has no clear winner — both approaches performed equally well in our limited setting.

Figure 14 shows the same experiment for the deterministic deque, only that our base machine is now one of the faster Pentium II processors. Again, by adding the slow hosts first the curve shows the typical improvement towards the largest experiment with 57 hosts. Obviously, this curve looks much less favorable, with a maximum speedup of only about 16 for 57 hosts. The bottom line, however, is that, even if a user has a powerful machine, our system can still speed up such an application by a considerable factor using large numbers of slower processors.

By varying the number of tasks, our second experiment shows that speedup improves when the hosts are better utilized. From the bar chart in Figure 15, we see that by selecting an adequate task size, and thus varying the number of atomic pieces, we significantly improve the speedup. With a task size of 32 x 32 pixels, yielding 1024 tasks, the speedup reached 23.7 for 32 hosts of type Pentium 166. Further reductions in task size did not result in improved speedup, only in increased communication. Speedup degraded to 14.98 for a task size of 16 x 16 pixels (4096 tasks). Increasing the size to 64 x 64 pixels (256 tasks) resulted in slightly slower per-

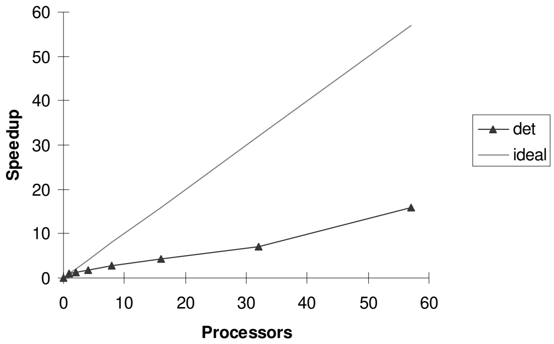

Figure 14: Practical Speedup for Deterministic Deque compared against a fast client.

formance than for size 32, although the difference between these two settings is hardly noticeable. In fact, on several runs the size 64 outperformed the size 32 (the chart reflects only our best results for each granularity). Finally, moving to 128 x 128 pixels (64 tasks) again resulted in degraded performance, this time due to load imbalances: since there are on average only 2 tasks per host, some hosts end up doing 3 tasks and slowing down the whole computation. The experiment clearly shows that the number of tasks should be relatively high with respect to the number of hosts, so that an even load can be achieved, but not unduly high so that communication is not excessive. This conforms well to what should happen in theory, and underlines that an application programmer should take the time to fine-tune the application before running it.

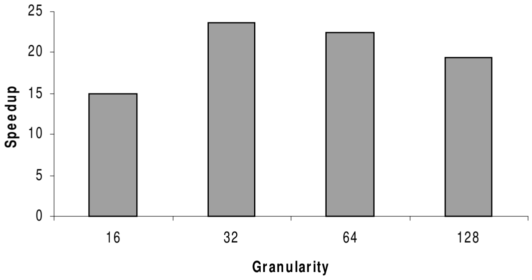

Figure 15: Different Granularities for the Deque.

Our final experiment compares the performance of the deterministic deque with and without the extra overhead of eager scheduling. These tests only reflect what happens in the *absence of failures*, not with actual host failures occurring. Since the only host slowed down by eager scheduling is the client that does the result processing, and result processing itself is very efficient, the overhead is generally insignificant, as can be seen in Figure 16. In light of the huge benefit of being able to make progress in the presence of failures, which are highly likely in the Internet setting, the small cost of the extra processing must be regarded as negligible.

To sum up, an image that took over 5 hours to render on a single computer took less than 7 minutes on 60 machines that were servicing undergrad and grad students, who unknowingly stole cycles from the image rendering as they surfed the web, emailed their friends, and occasionally compiled code.

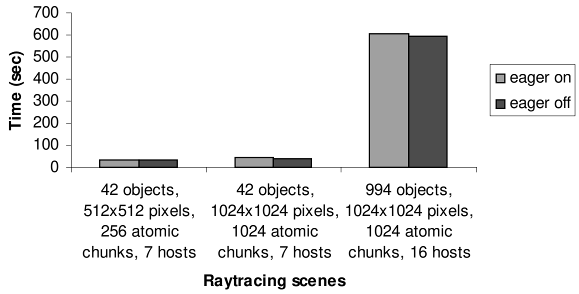

Figure 16: Eager Scheduling Overhead.

In the experiments for the original Javelin, the picture had more than 12 times as many objects, while the hosts (the Meiko's Sparc processors) were only about a fifth as fast, yielding tasks with a much larger computational load than the tasks used for these experiments. The communication speeds were about the same. We believe that, if our current experiments had a comparable computation/communication ratio, our speedups would be even closer to optimal. Also, the original Javelin experiments were run on an otherwise idle set of 64 processors (Meiko). Since the Meiko has been retired, and a dedicated multiprocessor machine is not the target environment for global computing, our Javelin++ experiments were run on a student laboratory whose machines were not idle, reflecting a realistic setting.

The tree built in the deterministic scheduler was *not* a bottleneck in these experiments; the time for a host to join the computation was about 70ms. We nonetheless are exploring the option of incorporating a new host into a waiting host tree as soon as it registers with the broker, potentially *before* any client requests service.

6 Conclusion

Parallel Internet computations need at least an order of magnitude more computers than conventional NOWs to justify their use. Global computing infrastructures thus must scale to at least an order of magnitude more computers than conventional NOWs. We have investigated the problem of creating a scalable, Java-based global computing infrastructure called Javelin++. We have presented an approach to distribute application code in a scalable manner through a network of brokers, each of which acts as a code cache for the hosts connected to it. We have also implemented two conceptually different approaches for managing a scalable computation: one that distributes work probabilistically, and another that distributes work deterministically. Our design analysis indicates that these versions of Javelin++ scale better than the original Javelin infrastructure. We believe both approaches will scale beyond what our experiments were able to verify. Our tests have confirmed the scheme's sensitivity to the computation/communication ratio. We thus hypothesize that as the computation/communication ratio increases, the speedups get closer to linear for much higher numbers of hosts. This hypothesis is not unreasonable; to achieve a computation/communication ratio comparable to that of a NOW, we must increase the computational load in the Internet setting to compensate for the increased communication time (relative to these times in NOWs).

In the future, we plan to focus on fault tolerance, which we see as a necessary precondition to achieve greater scalability; generalize our computational model, in order to accommodate any divide-and-conquer computation; organize hosts as soon as possible, potentially before a client requests service; contribute to the issues of correctness checking; and support host incentives.

References

[1] A. Alexandrov, M. Ibel, K. E. Schauser, and C. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] T. E. Anderson, D. E. Culler, and D. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1), Feb. 1995.

[3] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[4] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. An Infrastructure for Network Computing with Java Applets. *Concurrency: Practice and Experience*, 1998.

[5] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[6] D. Bhatia, V. Burzevski, M. Camuseva, G. Fox, G. Premchandran, and W. Furmanski. WebFlow–A Visual Programming Paradigm for Web/Java-based Coarse Grain Distributed Computing. *Concurrency: Practice and Experience*, 9(6):555–577, June 1997.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*, pages 207–216, Santa Barbara, CA, July 1995.

[8] V. Borda. Brycewarp. Master's thesis, Dep. of Computer Science, University of California, Santa Barbara, Santa Barbara, CA, 1998.

[9] N. Camiel, S. London, N. Nisan, and O. Regev. The POPCORN Project: Distributed Computation over the Internet in Java. In *6th International World Wide Web Conference*, Apr. 1997.

[10] P. Cappello, B. Christiansen, M. O. Neary, and K. E. Schauser. Market-Based Massively Parallel Internet Computing. In *Third Working Conf. on Massively Parallel Programming Models*, pages 118 – 129, nov 1997. London.

[11] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive Parallelism with Piranha. Technical Report YALEU/DCS/TR-954, Department of Computer Science, Yale University, New Haven, Connecticut, 1993.

[12] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, Nov. 1997.

[13] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[14] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997.

[15] G. Fox and W. Furmanski. Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modeling. *Concurrency: Practice and Experience*, 9(6):415–425, June 1997.

[16] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications — Confining the Wily Hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[17] Great Internet Mersenne Prime Search. GIMPS Discovers 36th Known Mersenne Prime. Press Release, Sept. 1997. http://www.mersenne.org/2976221.htm.

[18] A. S. Grimshaw, W. A. Wulf, and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), Jan. 1997.

[19] S. Hirano. HORB: Extended Execution of Java Programs. In *First International Conference on World-Wide Computing and its Applications (WWCA 97)*, 1997. http://ring.etl.go.jp/openlab/horb/.

[20] P. Kmiec. Consh: User-Level Confined Execution Shell. Master's thesis, Dep. of Computer Science, University of California, Santa Barbara, Santa Barbara, CA, Dec 1998.

[21] M. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[22] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, nov 1997.

[23] L. F. G. Sarmenta. Bayanihan: Web-Based Volunteer Computing Using Java. In *2nd International Conference on World-Wide Computing and its Applications*, Mar. 1998.

[24] L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Volunteer Computing Systems Using Java. *Future Generation Computer Systems Special Issue on Metacomputing*, 1998. To appear.

[25] D. W. Walker. Free-Market Computing and the Global Economic Infrastructure. *IEEE Parallel and Distributed Technology*, 4(3):60–62, 1996.