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: a probabilistic work-stealing scheduler and a deterministic scheduler. The deterministic scheduler also implements eager scheduling, as well as another fault-tolerance mechanism for 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 holding out the promise of 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 processors (NOW)s [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 global computing researchers. Matrix product also can be

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

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 is slow.

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, and presents two scalable designs for Javelin++: deterministic and probabilistic. 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.

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

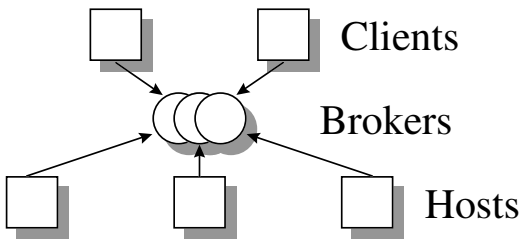

Figure 1: The Javelin Architecture.

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 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. Netscape also implements only a subset of JDK 1.1 functionality, although it supports RMI. As of today, the only agreed upon standard remains JDK 1.0.2.

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. Second, the ease of use property of Javelin is slightly weakened: Instead of simply pointing the web browser to a broker site, the user must now download the initial daemon class file, and then start a JVM that executes the daemon. This process is slightly more complex than before, but can be well explained on the broker's web page. Third, 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!

An alternative approach when Java applications are used would be to provide a special Javelin++ screen saver that the user could download and install on a host. Such a screen saver would run the JVM and the Javelin++ daemon while the host is idle, making the operation of Javelin++ convenient for the user. Although this has the disadvantages of making the installation harder for the user and having to provide a separate implementation for each OS, thus losing some of the platform independence of Java, it might be worth considering for the most popular operating systems[5].

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[6]. 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] MS Windows currently has more than 90% of the market.

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

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

1. Host allocation and code distribution — How does client find hosts for its computation, and how does t code get distributed efficiently to a potentially ve 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 3.2.1 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.

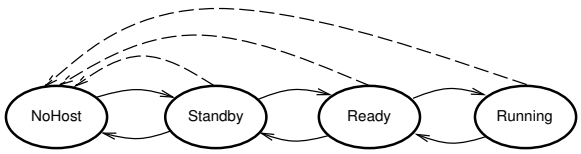

Figure 2: State Transition Diagram for Javelin++ Hosts.

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.

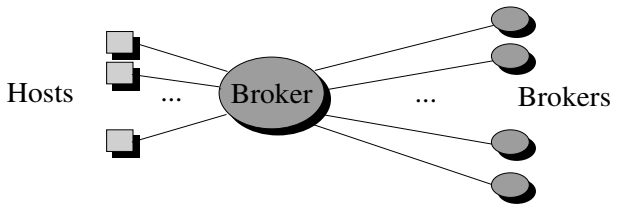

Figure 3: Broker Connections.

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[7]. 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 The Probabilistic Approach

In the probabilistic model we base our strategy on two main data structures that are local to every host: a *hash 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.

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

The task queue is double-ended because we follow the concept of *randomized work stealing* which was made popular by the Cilk project [7]. 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 will be processed. This means that when a host runs out of local jobs, it picks one of its neighbors at random from its hash table and issues a work request to that host. 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 his address table with the set of returned addresses. Thus, his knowledge of participants will increase until his 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.

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.

For fault tolerance purposes the next version of the deque will include *distributed eager scheduling*, where chunks of work can be reassigned to faster hosts in case results are still outstanding. Eager scheduling was made popular by the Charlotte project [5]. It is a low overhead way of ensuring progress towards the overall solution in the presence of failures or varying processor speeds.

3.2.2 The Deterministic Approach

The second version of Javelin++ implements a deterministic scheme. We chose to use a balanced tree — a heap — as the underlying structure of our deterministic model. As in the probabilistic approach, the fundamental concept employed is *work stealing* from a distributed deque. The main difference is that it follows a deterministic algorithm based on the tree structure.

Initially, each host retrieves a chunk 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 plus keeping the tree balanced ensures that each of the hosts gets a relatively even work load. The root of the tree is the client which is assumed to be a stable participant. When a new host joins, it is assigned a position at the bottom of the tree by the tree manager, which maintains

a heap data structure for that purpose. The tree fanout can be chosen individually for each application at startup.

The functionality of the tree itself becomes most important when a host fails or retreats. 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 traverses its ancestor chain until it finds a live ancestor. That host serves as the temporary parent of the host until the host has finished all work in its current deque. We thus avoid losing the work of the subtrees of a failed host. Once the chunk of work is finished, the host needs a new parent from which it requests new work. At this point, the host contacts the tree manager to get a new ancestor chain. If the empty position has

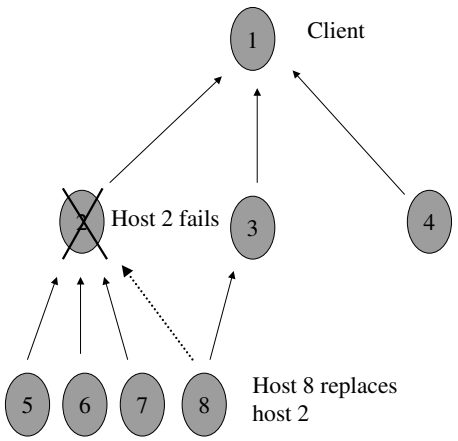

Figure 4: Deterministic tree

already been reported and filled, the tree manager traverses the tree representation, and returns a new ancestor list to 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 that it is to be moved. Figure 4 illustrates the situation where node 2 has failed and is replaced by node 8. Node 8 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. Currently, the tree is managed by one entity and therefore presents a potential bottleneck if the host failure rate is high. However, it would be possible to modify the existing implementation such that it would distribute the 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.

Although the results that we have obtained from both approaches are promising, we would like to examine a third approach—a synthesis of the probabilistic and deterministic versions. To further improve fault tolerance, we also plan to incorporate distributed eager scheduling in the next version of the deterministic deque.

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 specifying classes 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();
}
```

Any Javelin++ client must create a `JavelinClient` instance. The only constructor of `JavelinClient` takes the local hostname, the top-level classname used to load the host classes, and a broker's hostname. 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 of how the result actually propagates to the client depends on the underlying data manager. For instance, results could be sent directly to the client or collected and combined to be sent in larger chunks.

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 returnResult(Object result);
}

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

So far, we have not mentioned how the client specifies the work to a data manager. For this, the programmer has to write a class representing the type of work to be done which implements the `Splittable` interface, shown below. This way, the data manager has a means 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[8]. 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 object. 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.

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

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 register results. Once the data manager returns *null* or 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();

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

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

  public void returnResult(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 (true) {
      if ((Object work = dm.getWork()) == null)
        break;
      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 (true) {
  area = (RectEntry)dataManager.getWork();
  int [] pixels
       = tracer.raytrace(area.x, area.y,
                         area.width, area.height);
  dataManager.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 the new 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 deque and the deterministic tree, 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 run tests in a reasonable amount of time. The image is decomposed into a set of square sub-images, such as 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 the individual tasks 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, to force the tree to have some depth.

5.1 Measurements

Figures 5 and 6 show that, for both the deterministic and the probabilistic deque, the speedup degrades slightly from linear for the higher host numbers.

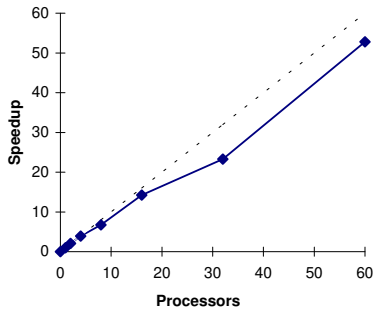

Figure 5: Speedup Curve for Random Deque.

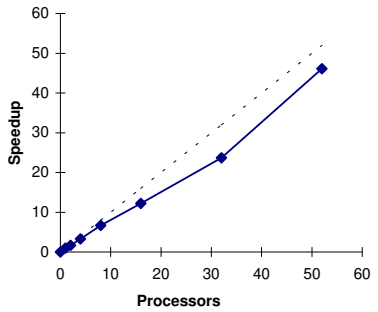

Figure 6: Speedup Curve for Deterministic Deque.

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 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.

By varying the number of tasks, experiment 2 shows that the speedup improves when the hosts are better utilized. From the bar chart in Figure 7, we see that by decreasing the task size, and thus increasing the number of tasks, we significantly improve the speedup. With a task size of 32 x 32 pixels, yielding 1024 tasks, the speedup reached 37.9 for 40 hosts. Further reductions in task size did not result in improved speedup, only in increased communication. Thus, an image that took over 5 hours (300+ min) to render on a single computer took less than 6 minutes on 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.

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 that our speedups would have been even closer to optimal. Also, the original Javelin experiments were run on an otherwise idle set of 64 processors (Meiko). Since such a setting with 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 situation much closer to actual reality.

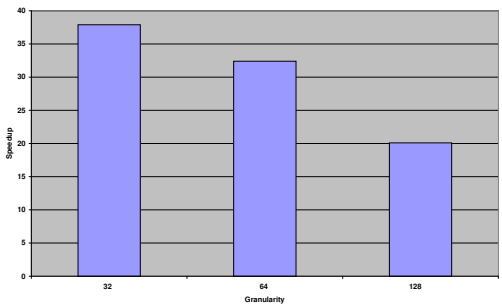

Figure 7: Different Granularities for the Deque.

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.

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 achived good results in a restricted setting, and we believe both approaches will scale beyond what our experiments were able

to verify. Our tests have have confirmed the scheme's sensitivity to 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 its increased communication time (relative to these times in NOWs).

In future, we plan to focus primarily on fault-tolerance, which will allow us to run experiments on a much larger scale. We also intend to generalize our computational model to accommodate any divide-and-conquer computation, 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.