Javelin 2.0: Java-Based Parallel Computing on the Internet

Michael O. Neary, Alan Phipps, Steven Richman, and Peter Cappello

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

Abstract. Javelin is a Java-based infrastructure for parallel Internet computing. This paper presents Javelin 2.0, focusing on the enhancements that distinguish it from prior versions of Javelin [9, 16, 15]. With regard to architecture, the paper presents enhancements that facilitate aggregating larger sets of host processors. The paper then presents: a branch-and-bound computational model, the supporting Javelin 2.0 architecture, a scalable task scheduler using distributed work-stealing, a distributed eager scheduler, implementing fault tolerance, and the results of performance experiments. Taken as a whole, Javelin 2.0 frees application developers from concerns about complex interprocessor communication and fault tolerance among Internetworked hosts. When all or part of their application can be cast as a piecework or a branch-and-bound computation, Javelin 2.0 allows developers to focus on the underlying application without sacrificing performance.

1 Introduction

Our goal is to harness the Internet's vast, growing, computational capacity for ultra-large, coarse-grained parallel applications. 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, 10, 1]. Some research projects that are designed to exploit this include *Charlotte* [5], *Atlas* [4], *Popcorn* [8], *Javelin* [9], *Bayanihan* [18], *Manta* [20], *Ajents* [11], and *Globe* [3]. Javelin 2.0 is designed to achieve 2 goals:

- Obtain the performance of a massively parallel implementation;
- Provide a simple API, allowing designers to focus on a recursive decomposition/composition of the parallelizable part of the computation.

Summarily, we want the application programmer to get the performance benefits of massive parallelism without the typically attendant costs: adulterating the application logic with interprocessor communication protocols, topology-specific (e.g., hypercube) interprocessor communication, and fault tolerance schemes. The resulting code should run well on as many processors as are available at any particular execution time, without any change to the program. Indeed, the program should perform well even when the set of processors *changes dynamically*.

In order to achieve these goals, Javelin 2.0 handles all interprocessor communication and fault tolerance for the application programmer, when the parallelizable computation can be cast as a branch-and-bound (or piecework) computation. This is a broad class of computations. We focus here on 2 issues that are fundamental to every application that is deployed as a global computation (i.e., a computation executing on a large set of Internetworked processors):

- *Scalable Performance* — If there is no niche where Java-based global computing outperforms existing multiprocessor systems, then there is no reason to use it. In order for global computing to outperform existing multiprocessor systems, it must harness a larger set of processors: The architecture must scale to a higher degree than existing multiprocessor architectures, such as networks of workstations (NOW) [2].
- *Fault tolerance* — An architecture that scales to thousands of hosts must be fault tolerant, particularly when hosts, in addition to failing, may dynamically disassociate from further participation in an ongoing computation.

Javelin 2.0 extends the piecework computational model to a branch-and-bound model which is implemented using a weak form of shared memory that itself is implemented via the *pipelined RAM* [12] model of cache consistency. This shared memory model is strong enough to support branch-and-bound computation (in particular, bound propagation), but weak enough to be fast. Using this cache consistency model, we present a high-performance, scalable, fault tolerant Internet architecture for branch-and-bound computations, such as are used to solve NP-complete problems, for example. For such an architecture to succeed, the architects must be diligently cognizant of the central technical constraint: On the Internet, *communication latency is large*.

The remainder of the paper is organized as follows: The next section presents the branch-and-bound model of computation. Section 3 presents the architecture of Javelin 2.0. It focuses on those enhancements that facilitate aggregating larger sets of hosting processors. Section 4 presents Javelin 2.0's task scheduler, shared memory, and fault tolerance scheme. Section 5 presents results of performance experiments, indicating remarkable speedups. The final section concludes the paper, outlining some immediately fruitful areas of global computing research and development.

2 Model of Computation

The branch-and-bound method intelligently enumerates all feasible points of a combinatorial optimization problem. By *intelligent* we mean that not all feasible solutions are examined. Branch-and-bound, in effect, produces a proof that the best solution is found without actually examining all feasible solutions. The method successively partitions the solution space (branches), and does not search a subspace (prunes), when there is sufficient information to infer that none of the subspace's solutions are as good as a solution (bound) that already has been found. (See Papadimitriou and Steiglitz [17] for a more complete discussion of branch-and-bound.) Here is a basic, sequential branch-and-bound algorithm:

```
activeset = {0}; // "0" is the original problem.
U = infinity;
while ( !activeset.empty() ) {
    node = activeset.select();   // removes node
    for (int i = 1; i <= node.numChildren; i++)
        if ( child[i].lowerBound() < U )
            if (child[i] is complete solution) {
                U = lowerBound[i];
                currentBest = child[i];
            }
            else
                activeset.insert( child[i] );
     // else child is killed implicitly
}
```

The computational model implies the following requirements:

– Tasks (elements of the activeset) are generated during the host computation
– When a host discovers a new best cost, it propagates it to the other hosts.
– Detecting termination in a distributed implementation requires knowing when all subspaces (children) have been either fully examined or killed.

The challenge, in sum, is to enable:

– hosts to create tasks, which subsequently can be stolen;
– hosts to propagate new bounds rapidly to all hosts;
– the eager scheduler to detect tasks that have been completed or killed

with a *minimum of communication*. The last bullet item is needed not just for termination detection, but also for fault tolerance: determining which tasks may need to be rescheduled.

The *piecework model of computation* [15] is a degenerate case of the branch-and-bound model, where:

– all tasks are generated initially (if only in bulk),
– no task is killed; bound distribution is unnecessary; termination detection is simplified.

To accommodate this new model, the 2.0 API required adding only a propagateValue method to the API [15].

3 Architecture

In this section we present the Javelin 2.0 system architecture. We begin with a brief overview of the participating entities and their responsibilities. Next, we explain the *Javelin Broker Name Service*, which serves as an entry point for any host willing to participate in the system. Finally, we give a brief description of our broker network topology and its host tree preorganization scheme.

3.1 Overview

The Javelin 2.0 system architecture retains the basic structure of its predecessors, Javelin [9] and Javelin++ [14]. 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. Figure 1 illustrates the Javelin 2.0 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.

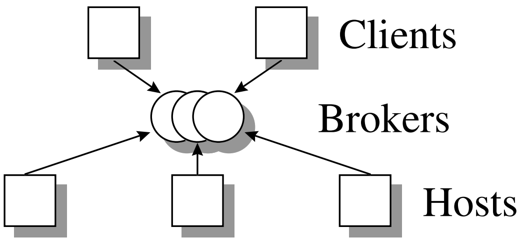

Fig. 1. The Javelin 2.0 Architecture.

3.2 Javelin Broker Name Service

When a host (or client) wants to connect to Javelin, it first must find a broker that is willing to serve it. The JavelinBNS system is a scalable, fault tolerant directory service that enables the discovery of a nearby Javelin broker, without any prior knowledge of the broker network structure. It is designed not only to aid hosts who are searching for brokers, but also to aid brokers who are looking for neighboring brokers.

A JavelinBNS system consists of at least two JavelinBNS servers[1]. Each server is responsible for managing a list of available brokers, responding to broker lookup requests, and ensuring that the other JavelinBNS nodes contain the same information. The JavelinBNS system thus serves as an information backbone for the entire Javelin 2.0 system.

Since the information stored for each broker is relatively small, the service will scale to a very large number of brokers. A small number of BNS servers will therefore be capable of administering thousands of broker entries, so a fully

[1] Information stored by a BNS server is fully replicated.

connected network of BNS servers will not be a bottleneck. At regular intervals, information is exchanged by the BNS servers. If a BNS server crashes and subsequently restarts, it can simply reload its tables with the information obtained from its neighbors, thus providing the necessary degree of fault tolerance.

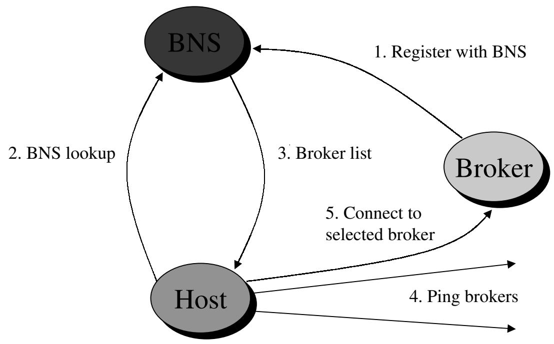

Fig. 2. JavelinBNS lookup sequence.

Figure 2 shows the steps involved in a broker lookup operation:

1. At startup, a broker registers its address with a known JavelinBNS server.
2. A host or client willing to participate in Javelin queries the BNS server for a list of k brokers for some constant k.
3. The BNS server *randomly* selects up to k broker addresses from its table and responds.
4. The host sends an RMI `ping()` call to each broker on the list.
5. The host evaluates the ping results and connects to the most suitable broker, provided the broker is willing to serve it. If not, it can pick the next broker or send another query to the BNS.

Likewise, brokers can themselves use the BNS to fill their address tables with neighboring brokers at startup. The list of known BNS servers is initially loaded from a configuration file, but can be updated by calling the BNS at runtime.

3.3 Broker Network & Host Tree 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. 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

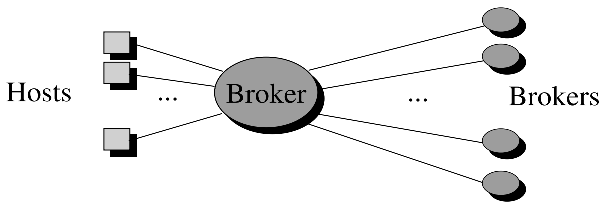

Fig. 3. Broker Connections.

degree of connectivity is higher than in a tree-based topology like the one used in the ATLAS project [4]. Figure 3 shows the connection setup of a broker.

When a host connects to a broker, the broker enters the host in a logical tree structure and responds with an RMI handle of the host's parent. The top-level host in the tree will not receive a parent; instead it will later become a child of the client. This way, the broker maintains a *preorganized tree of hosts* which are set on standby until a client becomes active. When a client connects, or client information is remotely received from a neighboring broker, the whole tree is activated in a single operation and the client information is passed to the hosts.

Brokers can individually set the branching factors of their trees, and decide how many hosts they can administer. In case of a host failure, the failed node is detected by its children and the broker restructures the tree in a heap-like operation (for details, see [15]). The additional burden of tree management was previously placed on the client, which could become a bottleneck for large trees. Thus, placing the tree management on the broker enhances scalability and increases performance when a client starts up, since the computation can begin immediately.

4 Scalable Computation & Fault Tolerance

4.1 The Scheduler

The fundamental concept underlying our approach to task scheduling 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 requests work from some host that it knows. One advantage of work stealing is its natural way of balancing the computational load, as long as the number of tasks is high relative to the number of hosts — a property well suited for adaptively parallel systems.

For our work stealing scheduler, 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. The deque is accessed 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 tasks, it selects a neighboring host from its address table, and requests work from that host.

Since the hosts are organized as a tree, the selection of the host to steal work from follows a deterministic algorithm based on the tree structure. Initially, each host retrieves work from its parent, and computes one task at a time. When a host finishes all the work in its deque, it attempts to steal work, first from its children, if any, and, if that fails, from its parent. This strategy ensures that all the work assigned to the subtree rooted at a host gets done before that host requests new work from its parent. Work stealing within a tree of hosts helps each host get a quantity of work that is commensurate with its capabilities. The client is the root of its tree of hosts.

4.2 Shared Memory

It might appear that we cannot implement shared memory efficiently among Internetworked processors in a manner that *scales*; the communication latency is too large. However, for branch-and-bound computation:

- only a *small amount* of shared memory is needed;
- a *weak shared memory model* suffices.

The small amount is because only one *int* or *double* is needed to represent a solution's *cost*. The weak model suffices because if a host's copy of best cost is stale, correctness is unaffected. Only performance may suffer — we might search a subspace that could be pruned. It thus suffices to implement the shared memory using a pipelined RAM (aka PRAM) model of cache consistency. This weak cache consistency model can be implemented with scalable performance, even in an Internetwork setting.

There are several methods to propagate bounds among hosts. We use the following: When a host discovers a solution with a better cost than its cached best cost, it sends this solution to the client. If the client agrees that this indeed is a new best cost solution (it may not be, due to certain race conditions), it updates its cached best cost solution, and "broadcasts" the new best cost to its entire tree of hosts. That is, it propagates the new best cost to its children, who in turn propagate it to their children, and so on, level by level down the host tree. This propagation is handled *asynchronously* by a separate *propagator thread*, avoiding the situation where a host blocks until all the hosts in its subtree have acknowledged the new bound. With a branching factor of 8, for example, 5 such "parallel" propagations can reach a total of 37,448 hosts. Another method, which we did not implement, is to use multicast.

4.3 Fault Tolerance

Eager scheduling *re*schedules a task to an idle processor in case its result has not been reported. It was introduced and made popular by the Charlotte project [5], and also has been used successfully in Bayanihan [19]. Javelin++ [15, 14] also uses eager scheduling to achieve fault tolerance and load balancing. It efficiently and relentlessly progresses towards the overall solution in the presence of host and link failures, and varying host processing speeds.

The Javelin 2.0 eager scheduler is located on the client. Although this may seem like a bottleneck with respect to scalability, it is not, as we shall explain below. Eager scheduling, however, is more challenging for branch-and-bound computation (as compared to piecework computation). Besides detecting *positive results*, (i.e., new best cost solutions), the eager scheduler must detect *negative results*: solution [subspaces] that have been examined and do not contain a new best cost solution, and solution subspaces that have been pruned. Performance, though, requires avoiding unnecessary communication and computation.

In a branch-and-bound computation, the size of the feasible solution space is *exponential* in the size of the input. In principle, the algorithm may need to examine all of these exponentially many feasible solutions to find the minimum cost solution. In practice, a partial solution, p, is "killed" (a subspace is pruned) when the lower bound on the cost of any feasible solution that is an extension of p *must be* more costly than the currently known minimum cost solution. The algorithm nonetheless must gather sufficient information to detect that the minimum cost solution has indeed been found. This implies that killed nodes and sub-optimal solutions must be detected by the eager scheduler. If a separate communication is required to detect each such event, the overall quantity of communication would nullify the benefits of parallelism. We cope with this communication overload by aggregating portions of the search space into atomic tasks, and similarly aggregating negative results into one communication per atomic task. This lets the eager scheduler know that this part of the problem tree has been searched, and hence need not be rescheduled. The number of negative communications consequently is equal to or less than the number of atomic tasks. In practice, it is much less than the number of atomic tasks; many are killed. We can adjust the computation/communication ratio by adjusting the size of atomic tasks, in order to decrease the overall run time. Performance is quite sensitive to atomic task size, so finding good size values is important.

For performance reasons, we balance the computational *size* of the hosts' atomic tasks with the client's computation of result handling and eager scheduling, so that neither the client nor the hosts have to wait for one another. Additionally, we want the number of atomic tasks to be much larger than the number of hosts, to keep them all well utilized, even when some are *much* faster than others. This lower bound on the size of the atomic task (to prevent the client from becoming a bottleneck) implies an upper bound on the number of atomic tasks we can create from a fixed size branch-and-bound computation. This upper bound, in turn, bounds the *number* of hosts that can be used effectively to solve the branch-and-bound computation. Thus, *decreasing* eager scheduling time *decreases* the minimum size of atomic tasks, which *increases* the number of atomic tasks, which *increases* the number of hosts that can be used effectively, which *decreases* the execution time.

A factor that pulls in the other direction of the preceding analysis concerns the relationship between eager scheduling and atomic task size: The *smaller* the atomic task size, the *more* atomic tasks there are, the *more* work must be done by the eager scheduler. In the final version of this paper we will plot, for a large, fixed problem size, the run time as a function of atomic task size. In light of these considerations, we expect the function to minimize somewhere in the middle of the domain of atomic task sizes.

In our present scheme, hosts communicate to the eager scheduler only when it has found a less costly solution than its cached minimum cost solution: Specifically, it communicates only the least cost solutions of atomic tasks that contain a solution that is less than its cached minimum cost solution. However, along with that communication, it conveys other atomic tasks that it has done, and nodes that it has killed. Thus, the number of communications necessary to provide sufficient information to detect termination is less than the number of nodes in the fringe of the tree, where the fringe is the set of nodes that represent either atomic tasks or killed partial solutions. Large tasks (i.e., tasks that are high in the problem tree) can be rescheduled. This results in hosts splitting these tasks, allowing the rescheduling and computation of their subtasks to be *distributed* (via work stealing) among the hosts.

The basic data structure required is a *problem tree*, which the client maintains to keep track of the computation status. The tree is constructed as atomic tasks report their local minimum cost solution. Each such atomic task is a leaf in this problem tree. The root of the problem tree represents the complete branch-and-bound computation. (In the TSP, it represents a starting vertex for a circuit in input graph.) Its children are the subproblems resulting from branching — a single split of the root problem. (In the TSP, children of the root problem represent a partial solution consisting of the first 2 vertices of a circuit in the TSP input graph.) This branching (splitting) continues, as we proceed down the problem tree: We subdivide it into smaller and smaller search spaces. (In the TSP, each node at level[i] in the problem tree represents a partial circuit consisting of the first i nodes of a circuit in the TSP input graph.) A parameter determines at what level splitting stops. At that point, a host will search the space for a solution that is less than the current minimum cost solution. (In the TSP, if the parameter is, say 10, then each atomic task is associated with a partial circuit consisting of the first 10 nodes of a circuit in the TSP input graph. Such a task is required to find the min cost circuit, if any, among those that start with those 10 vertices.)

As with the piecework model of computation, each node in the problem tree can be in one of 3 states: *done*, meaning the results for the subproblem have been received by the client; *partially 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.

In addition to the tree structure, undone nodes (tasks) are put in a circular linked list. Tasks are eagerly scheduled from this circular list until it becomes empty: there are no undone tasks. This indicates completion of the computation; the client then propagates a termination signal down the host tree. The processing itself consists of two distinct routines: In the description below, the task identifier "esTask" refers to a task object on the undone task list. It is that task that was most recently rescheduled.

```
public void processResult(Solution s){
  insert s into ProblemTree;
  mark s done;
  mark its ancestors in ProblemTree as either
       partially done or done, as appropriate;
  maintain undone task list;
}

public void selectTask(){
  // esTask refers to last eagerly scheduled node
  while ( (esTask = esTask.next() ) != null ){
    while ( esTask != null && !esTask.isAtomic() ){
      generate esTask's feasible children & their costs;
      insert children into ProblemTree;
      maintain undone task list;
      esTask = select one of these children;
    }
    if ( esTask.isAtomic() )
      // found atomic node to process
      return esTask;

    // else no feasible atomic node on this path of ProblemTree
  }
  done = true;  // set terminate signal
  return null;
}
```

There is another scheme for eager scheduling that we will consider. In this scheme, the eager scheduler infers for itself what nodes representing partial solutions are killed. This allows the discovery of killed nodes to be *deferred* until a task *must be eagerly scheduled*: There is an idle processor, and no work is available for stealing (i.e., no hosts have any atomic tasks that they themselves are not actively working on). The advantage of "lazy" killing — deferring the identification of killed nodes until no work can be stolen — is that, when killed nodes need to be identified, a tighter minimum cost bound is known, allowing the eager scheduler to kill tasks *higher* in the problem tree: less total work for the eager scheduler. A disadvantage is that only atomic tasks can be eagerly scheduled. Since hosts do not report killed nodes to the eager scheduler, if a task is rescheduled whose *sub*tasks are all killed, the eager scheduler would never receive this information, thus would eagerly schedule the task forever.

Our current implementation may communicate more killed nodes than ultimately is necessary because it communicates them sooner, when the minimum cost bound may not be so tight. However, our current implementation has the advantage that *non*-atomic tasks may be rescheduled, which enables the rescheduling and computation of subtasks to be distributed among the hosts. On the other hand, in the current implementation, a host does not report negative results and pruned subtrees until a new minimum cost solution is found. In the meantime, work that it has completed might be needlessly rescheduled. It thus is not clear to us which eager scheduling method would perform better overall for a given branch-and-bound computation, much less for most branch-and-bound computations.

5 Preliminary Experimental Results

Experiments were run in campus student computer labs under a typical workload for the network and computers. The heterogeneous test environment[2] consists of

- 4 Sun Enterprise 450 quad-processors with 256 MB RAM and a processor speed of 400 MHz,
- 14 Pentium II 350 MHz processors with 128 MB RAM,
- 14 Celeron 466 MHz processors with 128 MB RAM,
- 24 Pentium 166 MHz processors with 64 MB RAM, and
- a Beowulf cluster of 42 individual nodes, consisting of
  - 6 Pentium III 500 MHz quad-processors with 1 GB RAM, and
  - 36 Pentium II 400 MHz dual processors with 512 MB or 1 GB RAM.

The Beowulf cluster is running Red Hat Linux 6.0 on all nodes. All other machines are running Solaris 2.7 and are connected by a 100 Mbit network. We used JDK 1.2.1 with active JIT for our experiments.

We tested the performance of Javelin 2.0 with a TSP application. The performance was measured by recording the time to find the shortest tour in a given graph. The test graph is a complete, undirected, weighted graph of 22 nodes, with randomly generated integer edge weights w, $0 \leq w < 100$. This graph is complex enough to justify parallel computing, but small enough to enable us to run tests in a reasonable amount of time. The search tree is recursively decomposed by the splitting process described in Section 4 into atomic subtrees, which are then processed by a single host. The size of such an atomic subtree can be chosen freely by the application. For instance, if the splitting limit is set to 5, as in our experiments, any sub-path of 5 nodes from the root node will define a corresponding atomic subtree. For the given graph, this limit could potentially yield $21 \times 20 \times 19 \times 18 = 143,640$ atomic subtrees, although in a practical setting a large portion of these nodes will be eliminated by pruning.

We first measured the time it took a single processor to calculate the result. The test graph took approximately 8 hours to process on one of the slower

[2] This setup describes all possibly available machines. Actual testing only includes subsets of these, since, e.g., the cluster is still being configured, and not all of the other machines were ready for experiments either

machines, a Pentium 166. On a Sun E450 it took roughly 3.5 hours, and on our fastest processors, the Celeron 466s, it took just over 2 hours. The different base cases are shown in Figure 4.

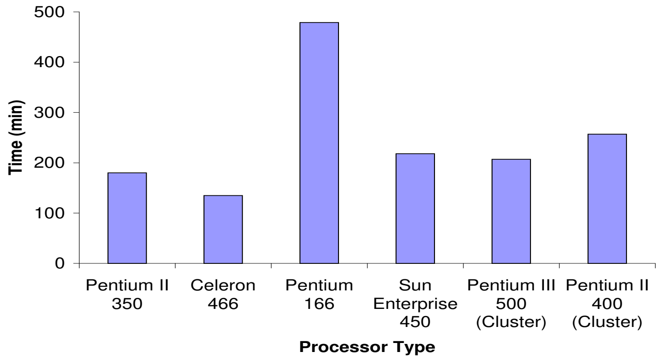

Fig. 4. Base Case Performance for TSP.

Host tree preorganization (see Section 3) significantly increases the performance at startup, as well as application scalability. We have successfully tested the host tree preorganization with as many as 4 brokers and 32 hosts. In a production environment, we would set the host 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.

We now discuss what we mean in our setting by the term "speedup". 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 if one tries to run tests in as homogeneous a hardware setup as possible, the varying workloads on both the OS and the network can amount 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 2.0 that is joined by a large set of hosts will definitely see "speedup"; the application will run faster than on a single machine. We will use the term *practical speedup* to distinguish between the two scenarios. In the following, we may omit the word "practical" when the meaning is clear from the context.

We now give a more formal definition of our notion of practical speedup: Let $M_1, M_2, ..., M_k$ denote k different processor types. Let $T_1(i)$ denote the time to complete the problem using 1 processor of type M_i. A conventional speedup, using p processors of type M_i can be defined as $T_1(i)/T_p(i)$. To compute speedup when we have more than one type of processor, we generalize this formula. Let a problem be solved concurrently using k types of processors, where there are p_i processors of type M_i: The total number of processors is $p = p_1 + p_2 + \cdots + p_k$. Let $T_p(p_1, p_2, \ldots, p_k)$ denote the execution time when using this mix of p processors. We define a *composite base* case that reflects this mix of processors:

$$T_1(p_1, p_2, \ldots, p_k) = \frac{p_1 T_1(1) + p_2 T_1(2) + \cdots + p_k T_1(k)}{p_1 + p_2 + \cdots + p_k}.$$

Finally, we define the speedup S as

$$S = T_1(p_1, p_2, \ldots, p_k)/T_p(p_1, p_2, \ldots, p_k).$$

While this definition does not incorporate machine and network load factors, it does reflect the heterogeneous nature of the set of machines.

Figure 5 shows the speedup we measured in our experiments and calculated according to the above formula. Speedup was superlinear in all measured configurations, topping out at a rather amazing 89.48 for 40 hosts. To observe superlinear speedup for the parallel TSP is quite common, due to the inherent irregularity of the input graph. For instance, say that the optimal tour is found by the sequential algorithm in one of the last pieces processed. If a parallel version finds the optimum much faster, it can spread the bound to all other hosts and they can prune the search tree much more efficiently, thus resulting in a much better running time and superlinear speedup. However, looking at our results it seems that the randomly generated test graph was *very* favorable to this kind of phenomenon.

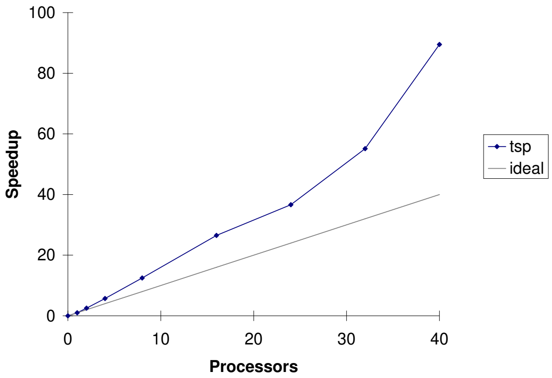

Fig. 5. Practical Speedup for TSP on Javelin 2.0.

To sum up, a graph that took almost 8 hours to calculate on a single computer took just over 4 minutes on 40 machines that were also servicing students with their normal, everyday workloads. We consider these results highly encouraging, although they need to be evaluated further with different input graphs and higher numbers of hosts[3]. Also, the effects of eager scheduling must be measured, since it may add significant overhead to the overall processing time. It will be interesting to see how varying the splitting limit affects performance. Being pushed for time, we could not incorporate these measurements in the current version of this paper. The final version will include these results.

6 Conclusion

The Internet has a vast ocean of computers. Aggregating those computers over relatively slow communication links in a way that is useful to massively parallel computation is an opportunity and a challenge. To attract application programmers, Javelin must:

- relieve the application programmer of the myriad technical details associated with using a dynamic set of Internetworked processors
- deliver essentially linear speedup over a large, *dynamic* set of hosts.

Javelin 2.0's API shields the application programmer from all such details, while obtaining essentially linear speedup.

To enlarge the set of applications that can benefit from Javelin, Javelin 2.0 extends Javelin++'s piecework model of computation to a branch-and-bound model. The technical challenge is to implement a kind of distributed shared memory that enables hosts to share the minimum cost upper bound, which is critical to pruning the search tree. The shared memory's principal implementation requirement is to rapidly and efficiently propagate a new bound from the host that discovered it to all other hosts. One's intuition suggests that distributed shared memory cannot be implemented efficiently among Internetworked processors in a scalable manner — communication latency is too large. We implemented the pipelined RAM model of cache consistency among hosts sharing the bound. Our experiments indicate that *limited* use of this weak shared memory poses *no* performance problem, even without resorting to multicast.

To further facilitate aggregating large numbers of hosts, Javelin 2.0 enhances host registration (with a broker): The host can request the broker name system to return a set of k broker names, where k is chosen by the host. Currently, the host then pings these brokers to discover the "nearest". We may implement an *expanding-ring search* [13], and compare its speed to our present implementation. Similarly, we may distribute code and initial data, and new bounds, to hosts using multicast, and compare its speed to that of our present implementation.

In Javelin 2.0, hosts organize into a tree under their broker. With but one Java RMI call on a broker, a client gets a handle to the broker's entire *preorganized* host tree. Other brokers convey their host trees with a similar economy of communication.

[3] We ran our tests at a bad time; over 20 of the machines were unavailable, and the new cluster facility with 96 processors is still being configured. In the next set of experiments we should be able to run on over 150 hosts!

Since parallel Internet computations need at least an order of magnitude more computers than conventional NOWs to justify their use, such infrastructures must scale to at least an order of magnitude more computers than conventional NOWs. But, using that many components indicates a fundamental need for fault tolerance. According to our goals for Javelin, part of our contract with the application programmer is to provide Java-based, high-performance network computing architecture that is both *scalable* and *fault tolerant*.

Regarding scalability, we were astonished by the speedups we observed in the TSP application. In the final version of this paper, we will run performance tests on several different random graphs and average the results. The TSP application appears to suggest that branch-and-bound computations can be sped up efficiently, even with large numbers of Internetworked hosts. Many combinatorial optimization versions of NP-hard problems are solved with branch-and-bound (e.g, the Integer Linear Programming problem). Perhaps others are well suited to Javelin 2.0.

Regarding fault tolerance, our distributed deterministic work-stealing scheduler integrates smoothly, not only with the caching scheme, but also with the distributed eager scheduler, which provides the essential fault tolerance (and contributes to load balancing). Javelin also detects and replaces hosts (interior nodes in the host tree) that have either failed or retreated from the computation.

In the future, we plan to 1) generalize our computational model, in order to parallelize any divide-and-conquer computation; 2) contribute to the fundamental issue of correctness checking; and 3) support host incentives, in order to attract a much larger set of computational hosts.

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. A. Bakker, M. van Steen, and A. S. Tanenbaum. From Remote Object to Physically Distributed Objects. In *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, Cape Town, South Africa, Dec. 1999.
4. 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.
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. 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.
9. 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.
10. 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.
11. M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *ACM 1999 Java Grande Conference*, pages 15–24, San Francisco, June 1999.
12. Lipton and Sandberg. PRAM: A scalable shared memory. Technical report, Princeton University: Computer Science Department, CS-TR-180-88, Sept. 1988.
13. T. A. Maufer. *Deploying IP Multicast in the Enterprise*. Prentice-Hall, 1998.
14. M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: Scalability Issues in Global Computing. In *ACM 1999 Java Grande Conference*, pages 171–180, San Francisco, June 1999.
15. M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: Scalability Issues in Global Computing. *Concurrency: Practice and Experience, to appear*, 2000.
16. M. O. Neary, B. O. Christiansen, P. Cappello, and K. E. Schauser. Javelin: Parallel Computing on the Internet. *Future Generation Computer Systems*, 15(5-6):659–674, Oct. 1999.
17. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.
18. L. F. G. Sarmenta. Bayanihan: Web-Based Volunteer Computing Using Java. In *2nd International Conference on World-Wide Computing and its Applications*, Mar. 1998.
19. L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, 15(5-6):675–686, Oct. 1999.
20. R. van Nieupoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Computing in Java. In *ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, June 1999.