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. This paper presents Javelin 2.0. It presents architectural enhancements that facilitate aggregating larger sets of host processors. It then presents: a branch-and-bound computational model, the supporting architecture, a scalable task scheduler using distributed work stealing, a distributed eager scheduler implementing fault tolerance, and the results of performance experiments. 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.

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 [1]. Some research projects that are designed to exploit this include *Charlotte* [4], *Atlas* [3], *Popcorn* [6], *Javelin* [7], *Bayanihan* [12], *Manta* [13], *Ajents* [8], and *Globe* [2]. Javelin 2.0 is designed to achieve two goals: 1) Obtain the performance of a massively parallel implementation; 2) Provide a simple API, allowing designers to focus on a recursive decomposition/composition of the parallelizable part of the computation. The application programmer gets the performance benefits of massive parallelism, without adulterating the application logic with interprocessor communication details and fault tolerance schemes. The resulting code should run well on a set of processors that changes during execution. 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 fundamental issues:

– *Scalable Performance* — If there is no niche where Java-based global computing outperforms existing multiprocessor systems, then there is no reason to use it. The architecture must scale to a higher degree than existing multiprocessor architectures, such as networks of workstations.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1231–1238, 2000.
© Springer-Verlag Berlin Heidelberg 2000

– *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 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* [9] 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 such an architecture to succeed, the architects must be diligently cognizant of the central technical constraint: On the Internet, *communication latency is large.*

2 Model of Computation

The branch-and-bound method, which generalizes the *piecework model of computation* [10], intelligently enumerates all feasible points of a combinatorial optimization problem: not all feasible solutions are examined. Branch-and-bound, in effect, proves that the best solution is found without necessarily examining all feasible solutions. The method successively partitions the solution space (*branches*), and prunes a subspace, when there is sufficient information to infer that none of the subspace's solutions are as good as a current solution (*bound*). (See Papadimitriou and Steiglitz [11] for a more complete discussion of branch-and-bound.) The computational model implies the following requirements: 1) Tasks (elements of the activeset) are generated during the host computation; 2) When a host discovers a new best cost, it propagates it to the other hosts; 3) Detecting termination in a distributed implementation requires knowing when all subspaces (children) have been either fully examined or killed. The challenge, in sum, is, with a *minimum of communication*, to enable: a) hosts to create tasks, which subsequently can be stolen; b) hosts to propagate new bounds rapidly to all hosts; c) the eager scheduler to detect tasks that have been completed or killed. The last item is needed not just for termination detection, but for fault tolerance, to determine which tasks need to be rescheduled.

3 Architecture

The Javelin 2.0 system architecture retains the basic structure of its predecessors, Javelin [7] and Javelin++ [10]. There are 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.

3.1 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 fully replicated JavelinBNS servers. 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 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 from its neighbors, thus providing for fault tolerance. Figure 1 shows the steps involved in a broker lookup operation.

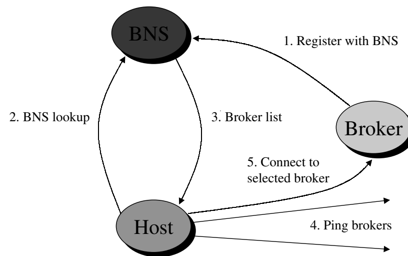

Fig. 1. JavelinBNS lookup sequence.

3.2 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. 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 degree of connectivity is higher than in a tree-based topology.

When a host connects to a broker, the broker enters the host in a logical tree structure. 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 [10]).

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 [5]. Work stealing is entirely demand driven — when a host runs out of work it requests work from some host that it knows. Work stealing balances 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.

In Javelin 2.0, tasks get split in a double-ended task queue 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 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 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

For branch-and-bound computation, only a *small amount* of shared memory is needed and a *weak shared memory model* suffices. The small amount is because only one integer 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 Internet 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, level by level down the host tree.

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 [4], and also has been used successfully in Bayanihan [12]. Javelin++ [10] 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.

5 Experimental Results

All experiments were run in campus computer labs under a typical workload. The heterogeneous test environment consists of 4 Sun Enterprise 450 dual/quad-processors with a processor speed of 400 MHz; 49 Celeron 466/500 MHz processors; and a Beowulf cluster of 42 nodes, with 6 Pentium III 500 MHz quad-processors, and 36 Pentium II 400 MHz dual processors. The cluster is running Red Hat Linux 6.0. All other machines are running Solaris 2.7. We used JDK 1.2 with active JIT for our experiments.

We tested the performance of Javelin 2.0 with a TSP application. The test graphs are complete, undirected, weighted graphs of 22 and 24 nodes, with randomly generated integer edge weights. These graphs are complex enough to justify parallel computing, but small enough to enable us to run tests in a reasonable amount of time. The 22-node graph took approximately 3 hours to process on a Sun E450. The 24-node graph took just under 10 hours on the same processor.

The term "speedup" is somewhat confusing here. Traditionally, speedup is measured on a dedicated multiprocessor, where all processors are homogeneous in hardware and software configuration, and varying workloads between processors do not exist. Thus, 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 performance of hosts. However, from a practical standpoint, a user running an application on Javelin 2.0 with 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 $M_1, \ldots, M_k$ denote k different processor types. Let $T_1(i)$ denote the time to complete the problem using 1 processor of type M_i. 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 $p = p_1 + \cdots + p_k$. Let $T_p(p_1, \ldots, p_k)$ denote the execution time when using this mix of p processors. We define a *composite base* case that reflects this mix of processors:

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

Finally, we define the speedup S as

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

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

Figure 2 shows the speedup we measured in our experiments and calculated according to the above formula. For the 22-node graph, speedup was superlinear at first, until it topped out at 77.26 for 100 hosts, when communication became a significant bottleneck. To observe superlinear speedup for the parallel TSP is quite common, due to the inherent irregularity of the input graph. The results for the 24-node graph illustrate this even further. Here, speedup was nowhere near as good, reaching only 23.35 for 80 hosts. However, the curve still shows a steady rate of improvement, and the larger graph has the potential to scale better due to its higher computation complexity.

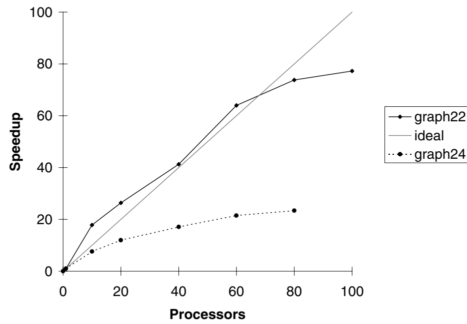

Fig. 2. Practical Speedup for TSP on Javelin 2.0.

To sum up, a graph that took about 3 hours to calculate on a single computer took just under 3 minutes on 100 processors under their normal workloads. These results are encouraging, although they need to be evaluated with different input graphs and more hosts.

6 Conclusion

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 distributed shared memory that enables hosts to share bounds. 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.

To facilitate aggregating large numbers of hosts, Javelin 2.0 enhances host registration: The host can request the broker name system to return k broker names, where k is chosen by the host. Currently, the host then pings these brokers to discover the "nearest". In Javelin 2.0, 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.

The TSP experiments suggest that branch-and-bound 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.

Our distributed deterministic work stealing scheduler integrates smoothly, not only with bound caching, but also with the distributed eager scheduler, which provides essential fault tolerance.

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] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Lipton and Sandberg. PRAM: A scalable shared memory. Technical report, Princeton University: Computer Science Department, CS-TR-180-88, Sept. 1988.

[10] 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.

[11] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[12] 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.

[13] 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.