JANET: A Java-Centric Network Computing Service

Peter Cappello and Christopher James Coakley
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA, USA 93106
email: {cappello | ccoakley}@cs.ucsb.edu

Abstract

JANET is a Java-centric distributed service for high-performance parallel computing. The paper describes JANET's API, which is especially suitable for computations that can be cast in the divide-and-conquer paradigm. Computational tasks have access to a global (read-only) input object and a mutable object that is shared with best-effort. These features enable an elegant expression of branch-and-bound optimization, which is used as the benchmark computation in the performance experiments. The API includes a simple set of application-controlled directives for improving performance by reducing communication latency or overlapping it with task execution. The architecture manages a host processor set that can change during the program execution for reasons that include faulty hosts. Preliminary experimental results confirm high parallel efficiency on a branch-and-bound implementation of the Traveling Salesman Problem, applied to a randomly-generated 150-city problem instance.

INTRODUCTION

Most distributed computing systems that harvest idle cycles do so for problems that admit a master-worker algorithm that is simple in at least two respects:

- Each task's computational load is known at compile time, possibly as a function of data parameters (e.g., the number of pixels to be rendered in a raytracing task);
- Worker tasks do not communicate among themselves. They merely report their result to the master process.

Such algorithms can be found for many important problems (e.g., protein folding, genetic sequencing, parameter sweep optimization problems). The number of distributed systems that support this kind of algorithm is growing rapidly. For example, there is SETI@home [5] from UC Berkeley, the *fight against cancer* [3] from Parabon Computation, *folding@home* [1] from Stanford for studying protein folding,

screensaver lifesaver [4] from Oxford in collaboration with United Devices that searches for anti-cancer drugs, and *fight AIDS at home* [16] from Entropia.

There is increasing interest in investigating *more complex* algorithm classes for cluster/network/grid computing:

- where compute servers may join and/or leave during program execution;
- that tolerate faulty compute servers during program execution;
- that that scales to a large number of computer servers with good speedup.

Branch-and-bound constitutes a large class of algorithms. The simple Master-Worker algorithms, described above, form a degenerate case. Branch-and-bound's implementation is more complex than master-worker, since:

- The fixed-depth task decomposition tree associated with Master-Worker generalizes to a dynamic-depth task decomposition tree.
- The task decomposition tree is quite unbalanced, due to the bounding process (also known as pruning), in a way that depends on the problem instance, and thus is revealed only at execution time.
- Tree pruning, to proceed efficiently, requires communication among the compute servers, as they discover new bounds.

A wealth of important combinatorial optimization problems are routinely solved via branch-and-bound (e.g., Integer Linear Programming, the Traveling Salesman Problem, and combinatorial auction winner selection [32], and a variety of operations research problems). Thus, if we can *efficiently speed up* an adaptively parallel implementation of branch-and-bound computations on a distributed system that tolerates faulty compute servers, then the usefulness of, and demand for, cluster/network/grid computing will increase dramatically. Indeed, some work has been done on fault-tolerant, distributed branch-and-bound [41, 28, 31, 29, 30].

The principle contribution of this paper to the study of adaptively parallel cluster/network/grid computing is to integrate architectural and language constructs that enable the simple expression of branch and bound computation, and a distributed implementation that 1) tolerates faulty compute servers, 2) distributes termination detection, and 3) exhibits measurably good speedup.

Background

A few years ago, a pair of developments ushered in a new era. One was the announcement of the SETI@home [5] project. Since then, similar projects have arisen (see above). While such applications are of the simple Master-Worker variety, compute hosts in this system work on self-contained pieces of a large problem, it they promote a vision of a world-wide virtual supercomputer [21, 6] or globally distributed computations.

The related development was an emerging vision of a computational *Grid*, an "integrated, collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by

multiple organizations." [20]. The Globus project [20] is the best known manifestation of that vision. In contrast to Globus, which is open source, Avaki [24] is a proprietary version of that vision, rooted in Legion research [25], as is Sun's GridEngine (http://www.sun.com/grid/) [2]. comes in both free and non-free versions. Such systems are not Java-centric, and indeed *must be* language-neutral. The Millennium Project [17] is a "system of systems" project where a key element of the research is how constituent systems interact. The Ninja project [40] is developing an infrastructure for Internet-based *services* that are scalable, fault tolerant, and highly available.

Some application development or deployment systems are *explicitly* based on the grid, such as AppLeS for parameter sweep computation by Casanova et al. [15], and Condor-G by Frey et al. [22] (where Condor [18] makes use of several grid services, such as GSIFTP). EveryWare [41] pushed this envelope by presenting an application development toolkit that was demonstrated to "leverage Globus, Legion, Condor, NetSolve Grid computing infrastructures, the Java language and execution environment, native Windows NT, and native Unix systems in a single, globally distributed application"[41]. In such an environment, the experiment ran on compute resources that were shared by other applications. Thus, measuring speedup is problematic, and was not done. This is unfortunate for us, since the application, a Ramsey number search, was implemented as a branch-and-bound problem. Using the toolkit to develop an application that interfaces with this disparate set of components is reputed to require extremely broad expertise. The explicit goal of the GrADS project is to *simplify* the development and performance tuning of such distributed heterogeneous applications destined for the Grid [26].

There is a growing niche of coarse-grain, parallel applications for which the Java programming system is a suitable environment. Java's attraction includes its solution to the portability/interoperability problem associated with heterogeneous machines and OSs. Please see [38] for an enumeration of other advantages. The use of a virtual machine is a significant difference between Java-based systems and previous systems. Typically, the Java niche includes new applications, which thus do not need to interface with legacy code. These Java-based efforts are orthogonal to the Grid work: In principle, they can be loosely coupled to the Grid via Grid protocols: *Advances in one can be leveraged by the other*. The Java CoG Kit[38] facilitates such leveraging activity. The Java-based research can be partitioned into:

1. systems that run on a processor network whose extent and communication topology are known *a priori* (although which particular resources are used to realize the processor network may be set at *deployment time*, perhaps via Grid resource reservation mechanisms);

2. systems that make no assumption about the number of processors or their communication topology.

Category 1 includes, for example, Java MPI systems and applications. JavaSymphony [19], also in this category, is an extensive pure Java class library constituting a high-level API that enables the application programmer to direct the exploitation of the application's parallelism, the physical system's locality (represented as a virtual hierarchy of physical compute nodes), and load-balancing based on both static and dynamic properties (e.g., storage capacity and available memory). *Manta* [36], another example, elegantly shows that wide-area networks, modelled as two-level networks, can efficiently support large, coarse-grain parallel computation. Ibis [37] is, in a sense, an outgrowth of the Manta research. It is a programming environment that is portable (it is pure Java), robust in the face of dynamically available networks and processors (although it does not support faulty processors *during* an execution), and has efficient, object-

based communication. Its flexibility and efficiency derive from its implementation on top of a carefully designed Ibis Portability Layer, a set of interfaces whose implementation can be plugged in at runtime.

Systems of category 2 can be further subdivided:

- those that support adaptive parallelism (originally defined in [23]);
- those that do not.

The second of these categories includes, for example, the system by Casanova et al. for *statically* scheduling parameter sweep computations on a set of resources determined a priori via the Grid). The first category includes Popcorn [11], Charlotte [8], Atlas [7], Javelin, and Bayanihan [33], among others. Popcorn [11] was the first Internet-based Java-centric project, as far as we know, that explicitly used market mechanisms (e.g., Vickrey auctions) to motivate "sellers" of computational capacity. Charlotte used eager scheduling, introduced by the Charlotte team, and implemented a fully distributed shared memory. It was designed for a local area network (LAN). ParaWeb [10] emulates a fully functional shared memory parallel computer, and hence also is best suited to a LAN. Atlas is a version of Cilk-NOW intended for the Internet setting. As a master's project, it terminated abruptly, and, in our opinion, without reaching its full potential: testing never exceeded eight processors. Bayanihan also uses eager scheduling. Its initial implementation however does not scale to large numbers of processors. Like Atlas, CX [13, 14] supports a divide-and-conquer API, and adaptively parallel algorithms with faulty processors.

Within this informal taxonomy, JANET is Java-centric, tolerates faulty compute servers, and supports small-grained, adaptively parallel computation with a divide-and-conquer API augmented by a mutable object that is shared with best effort. It is compatible with large-scale cluster settings, LAN/WAN, and corporate intranets.

COMPUTATIONAL MODEL

Computation is modelled by an *acyclic directed graph* (ADG) whose nodes represent tasks. An arc from node v to node u represents that the output of the task represented by node v is an input to the task represented by node u. Such a ADG graph is illustrated in Fig. 1.

Tasks have access to a *shared object*. The semantics of "shared" reflects the envisioned computing context—a computer network: The object is shared with best effort. This limited form of sharing is of value in only a limited number of settings. However, branch and bound is one such setting, constituting a versatile paradigm for coping with computationally intractable optimization problems.

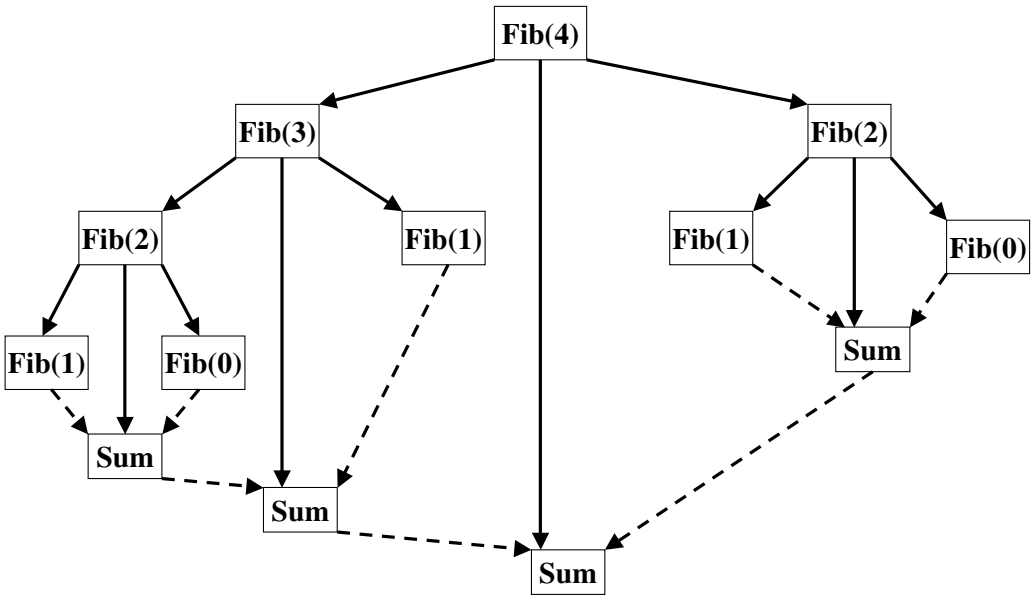

Figure 1: A Task graph for doubly recursive computing of the 4th Fibonacci number.

API

Divide and conquer

Again, the programmer views the computation as an acyclic directed graph (ADG), whose nodes represent computational tasks. An arc from node A to node B indicates that the output of the task represented by node A is an input to the task represented by node B. Tasks thus are central to the API because they encapsulate computation. Each task has an array of input objects (possibly of length 0). It either produces an output object or decomposes into a sequence of subtasks and an associated *compose* task. The function of the compose task is to compute an output from the subtasks' outputs. The public interface for Task is as follows:

```
    public void   compute( Task subtask );
    public Object execute( Environment environment );
    public Object getInput( int i );
    public int    numInputs();
```

We illustrate its use with the proverbial "Hello world!" of divide-and-conquer: A recursive Fibonacci number calculation.

```
final public class F extends Task {
    private int n;
    public F( int n ) { this.n = n; }

    public Object execute ( Environment environment ) {
        if ( n < 2 ) {
            return  new Integer( 1 );
        } else {
            compute( new F( n - 1 ) );
            compute( new F( n - 2 ) );
            return new Sum();
}   }   }

public final class Sum extends Task {
    public Object execute ( Environment environment ) {
        int sum  = ((Integer) getInput( 0 )).intValue();
            sum += ((Integer) getInput( 1 )).intValue();
        return new Integer( sum );
}   }
```

In this example, the F task returns an output—an Integer, if $n < 2$, but decomposes the problem into 2 F subtasks whose output is composed into F's output via the the Sum task. The Sum task, as defined, never decomposes into subtasks. In the F task, the subtasks are first constructed, then "dispatched" via Task's compute method. Thus, the ADG of tasks is revealed only at execution time.

The common environment

This example does not use the Environment, which we now describe. This class has the following public interface.

```
    public void   fetchTask();
    public Object getInput();
    public Shared getShared();
    public void   setShared( Shared shared );
```

Environment is used to present a common environment to tasks. The Environment constructor takes initial values for an input object and a shared object. The input object has data that any task may read via the getInput method. For example, in a traveling salesman problem (TSP), one might want the distance matrix to be in the Environment's input object.

Tasks access the shared object via the Environment getShared method, and can mutate it via the setShared method. A shared object implements the `Shared` interface:

```
public interface Shared extends java.io.Serializable {
    public Object get();
    public boolean isNewerThan( Shared shared ) throws NullPointerException;
}
```

The semantics of shared are weak, but appropriate for distributed computing: When the value of a Shared object changes, its value is propagated to with best effort. When an environment receives a new value for the shared object, it asks the shared object if the proposed new value is indeed newer than the current value, via the isNewerThan method: The *implementation* of the Shared object operationally defines the meaning of "newer." For example, in a TSP problem, the distance of the best known tour may be a shared object. Then, when a new tour is found, its total distance can be shared. In this case, the isNewerThan method would return true if and only if the proposed total distance is, in fact, less than the total distance of the currently best known tour. In a distributed setting this may not always be the case, hence the need to check. The IntUpperBound class illustrates these ideas:

```
public class IntUpperBound implements Shared {
    private Integer shared;
    public IntUpperBound(int shared ) { this.shared = new Integer( shared ); }
    public synchronized Object get() { return shared; }
    public synchronized boolean isNewerThan( Shared shared ) {
        if ( shared == null ) {
            throw new NullPointerException();
        }
        return this.shared.intValue() < ( (Integer) shared.get()).intValue();
}  }
```

We illustrate the use of the shared object, using the TSP example. The MinSolution class takes some number of Solution objects, and returns that solution object of least total distance, if its total distance is less than the total distance in its shared object. Notably, if it finds a solution that is better, it updates the shared object, using the setShared method.

```
public final class MinSolution extends Task {
    public Object execute ( Environment environment ) {
        Solution currentBest = null;
        for ( int i = 0; i < numInputs(); i++ ) {
            Solution reportedSolution = (Solution) getInput( i );
            if ( reportedSolution != null ) {
                int lowerBound = reportedSolution.getLowerBound( environment );
                if ( lowerBound <= sharedUpperBoundValue( environment ) ) {
                    environment.setShared( new IntUpperBound( lowerBound ) );
                    currentBest = reportedSolution;
        }   }   }
        return currentBest;
    }

    private int sharedUpperBoundValue( Environment environment ) {
        return ( (Integer) environment.getShared().get() ).intValue();
}   }
```

Also, MinSolution's execute method invokes numInputs, the Task method that returns the number of inputs to *this* task (which are not to be confused with the Environment input object that is common to *all* tasks).

The application programmer is not shielded from the dominating physical constraint on networked computation among compute servers whose availability may be short-lived: *Communication latency is large relative to processor cycle time.* The fetchTask method enables the programmer to impart performance information to the Task scheduler: A task can request that another task be pre-fetched. This may be desirable when a task does not decompose into subtasks. In that case, while one task is being executed, another task is being fetched. Overlapping computation with communication hides latency. The latency problem, intrinsic to distributed computing, is exposed to the application programmer, who oftentimes knows how best to cope with it *for his/her application*. We continue our discussion of task pre-fetching when we discuss task caching, another important latency hiding technique in JANET.

ARCHITECTURE

JANET, a JAva-centric NETwork computing service that supports high-performance parallel computing, is an ongoing project that:

- virtualizes compute cycles;
- stores/coordinates *partial* results - supporting fault-tolerance;
- is partially self-organizing;
- may use an open grid services architecture [34, 35] frontend for service discovery (not presented);

- is largely independent of hardware/OS;
- is intended to scale from a LAN to the Internet.

JANET is designed to: **support scalable, adaptively parallel computation** (i.e., the computation's organization reduces *completion* time, using many *transient* compute servers, called *hosts*, that may join and leave during a computation's execution, with high system efficiency, regardless of how many hosts join/leave the computation); **tolerate basic faults**: JANET must tolerate host failure and network failure between hosts and other system components; **hide communication latencies**, which may be long, by overlapping communication with computation.

JANET comprises 3 service component classes.

Hosting Service Provider (HSP): JANET clients (i.e., processes seeking computation done on their behalf) interact solely with the hosting service provider component. A client logs in, submits computational tasks, requests results, and logs out. When interacting with a client, the hosting service provider thus acts as an agent for the entire network of service components. It also manages the ensemble of service components. For example, when any service component wants to join the distributed service, it first contacts the hosting service provider. If the component is a task server, the HSP tells the task server where it fits in the task server network; if it is a host, the HSP tells the host which task server it is assigned to. When a host fails, its task server notifies the HSP of this fact.

Task Server: This component is a store of Task objects. Each Task object that has been spawned but has not yet been computed, has a representation on some task server. Task servers balance the load of ready tasks among themselves. Each task server has a number of hosts assigned to it. When a host requests a task, the task server returns a task that is ready for execution, if any are available. If a host fails, the task server reassigns the host's tasks to other hosts.

Host: Each host repeatedly requests a task for execution, executes the task, returns the results, and requests another task. It is the central service component for virtualizing compute cycles.

When a client logs in, the HSP propagates that login to all task servers, who in turn propagate it to all their hosts. When a client logs out, the HSP propagates that logout to all task servers, which *aggregate* resource consumption information (execution statistics) for each of their hosts. This information, in turn, is aggregated by the HSP for each task server, and returned to the client. Currently, the task server network topology is a torous. However, scatter/gather operations, such as login and logout, are transmitted via a task server *tree*: a subgraph of the torous. See Figure 2.

Task objects *encapsulate* computation: Their inputs & outputs are managed by JANET. Task execution is idempotent, supporting the requirement for host transience and failure recovery. Communication latencies between task servers and hosts are reduced or hidden via task caching, task pre-fetching, and task execution on task servers for selected task classes.

Task servers bind to the service as they become available. Hosts bind to, and unbind from, task servers as they become [un]available, *not* when a client requests service. This enables a rapid response to client service requests, pursuant to JANET's goal of being a high-performance parallel computing service. The time for host [dis]aggregation is not charged to the client; it is managed according to host availability,

which is presumed to be transient. This is in contrast to distributed computing approaches where resource aggregation is client-initiated [27]. The difference has to do with a difference of requirements. Ours is to support high-performance parallel computing: The time to respond to a client request must be minimal: We avoid client-initiated aggregation in pursuit of minimal *completion times* for our clients.

SCALABILITY

System scalability implies that memory, computation rate, and bandwidth requirements of system components does not grow as a the number of such components gets large. JANET achieves this for its host and task server component classes. The number of hosts that can be serviced by an individual task server is bounded, which also bounds the task server's associated memory and bandwidth requirements. A task server's computation rate requirements are similarly bounded by a constant with respect to the number of system components. However, Task scheduling requires time that is $O(\log n)$ where n is the number of *ready* tasks: Task server computation time for a Task request is a slow-growing function of the number of ready tasks currently held by the task server, which in turn depends on the *client application*. The torous network of task servers scales: The number of task server network connections maintained by a single task server is less than or equal to 4, regardless of how many task servers are in the network.

Apart from [un]registration, the HSP communicates with only the "root" task server. The memory requirement for the HSP is linear in the number of task servers and logarithmic in the number of hosts. A current theoretical bottleneck concerns host registration. Since every host must first register with its HSP, the HSP could, in principle, become a bottleneck if the [un]registration rate goes above the HSP's ability to respond. We expect that an HSP can support at least 100 [un]registrations per second (we have not measured the maximum registration rate). Since we expect that hosts will be available, on average, for several minutes, we expect that an HSP can support an assemblage of many thousands of transient hosts. In situations where hosts are even more stable, the HSP clearly does not constitute a host [un]registration bottleneck. Task servers also register with their HSP. But, task servers are much more stable than hosts. Hence, task server [un]registration does not present a scalability issue for the HSP: Host [un]registration rates would become a problem long before task server [un]registration rates would.

TOLERATING FAULTY HOSTS

To support self-healing, all proxy objects are leased (a basic concept in the Jini architecture). When a task server's lease manager detects an expired host lease and the offer of renewal fails, the host proxy: 1) returns the host's tasks for reassignment, and 2) is deleted from the task server.

HIDING COMMUNICATION LATENCY

JANET's API includes a simple set of *application-controlled* directives for improving performance by reducing communication latency or overlapping it with task execution.

Task caching When a task constructs subtasks, the first constructed subtask is cached on the host, obviating the need to ask the TaskServer for its next task. Thus, *the application programmer controls which task, if any, is cached.*

Task pre-fetching The API encourages the programmer to focus on the task structure of the computation. However, the programmer *does* need to be concerned with communication latency. The application can help hide communication latency via task pre-fetching: When a task knows that it is not going to construct any subtasks, it can invoke its environment's prefetch method. This causes the host to request a task from the task server in a separate thread. The host's execution of the current task thus overlaps with its request for the next task. The programmer is thus motivated to 1) separate task classes that construct subtasks from those that do not (which then immediately invoke prefetch), and 2) constitute non-decomposable tasks with compute time that is at least as long as a Host—TaskServer round trip (on the order of 10s of milliseconds, depending on the size of the returned task, which affects the time to marshal, send, and unmarshal it).

Task server computation When a task's encapsulated computation is little more complex than reading its inputs, *the task executes on the task server*. This is because the time to marshal and unmarshal the input plus the time to marshal and unmarshal the result is less than the time to simply compute the result (not to mention network latency). Thus, a task server computes selected Task classes, such as binary boolean operators, min, max, sum (typical linear-time gather operations). (Currently) if a task class extends Compose, its task objects execute on the task server: *The application programmer controls the use of this important performance feature.*

Taken together, these features reduce or hide much of the delay associated with Host—TaskSever communication.

PERFORMANCE EXPERIMENTS

THE TEST ENVIRONMENT

We ran our experiments on a Linux cluster. The cluster consists of 1 head node, and 32 compute nodes. Each node is a dual 2.6GHz Xeon processor with 3GB of PC2100 memory, two 36GB SCSI-320 disks with on-board controller, and an on-board 1 Gigabit ethernet adapter. The machines are connected via the gigabit link to one of 2 Asante FX5-2400 switches. Each node is running Red Hat 8 with the Linux smp kernel 2.4.18-27.7.xsmp, and the Java j2sdk1.4.2.

THE TEST PROBLEM

We ran a branch-and-bound TSP application, giving it a randomly-generated 150-city problem. To ensure that the speedup could not be super-linear, we set the initial upper bound for the minimal-length tour with the optimum tour length. Consequently, each run explored exactly the same search tree: Exactly the same set of nodes is pruned regardless of the number of parallel processors used. Indeed, the problem instance decomposes into exactly 16,023 BranchAndBound tasks whose total execution time was 23,151.74 seconds (for an average task time of 1.45 seconds), and exactly 8,011 MinSolution tasks whose total execution time was less than 8 seconds (with an average task time of less than 1 millisecond).

THE MEASUREMENT PROCESS

The experiment was run using 2 builds of Janet, identical in all respects except one: One build, hard coded the number of Janet host "task processors" to 1, in order to compute the 1-processor base case. The other build, used in all other tests, set the number of host "task processors" to the number of available processors. For each experiment, a hosting service provider was launched, followed by a single task server on the same node. Each host was started on its own node. Except for the base case, each host thus had access to 2 processors. After the JANET system was configured, a client was started on the same node as the HSP (and task server), which executed the application. The application consists of a *deterministic workload* on a very unbalanced task graph. Measured times were recorded by Janet's *invoice system*, which reports elapsed time (wall clock, not processor) between submission of the application's source task (aka root task) and receipt of the application's sink task's output.

THE MEASUREMENTS

T_P denotes the time for P physical processors to run the application. A computation's *critical path time*, denoted T_∞, is a maximum time path from the source task to the sink task. We captured the critical path time for this problem instance: It is 64 seconds. A well known *lower* bound on the completion time [9] is $\max\{T_\infty, T_1/P\}$. Thus, $(\max\{T_\infty, T_1/P\})/T_P$ is a *lower bound* on the fraction of perfect speedup that is actually attained. Figure 3 presents speedup data for several experiments: The ordinate in the figure is the *lower bound* of fraction of perfect speedup. As can be seen from the figure, in all cases, the actual fraction of perfect speedup exceeds 0.90. Specifically, the 1-processor base case ran in 6 hours and 18 minutes; whereas the 52-processor experiment (2 processors per host) ran in just 8 minutes!

We are very encouraged by these measurements, especially considering the small average task times. Javelin, for example, was not able to achieve such good speedups for such short tasks. Even CX [13, 14] is not capable of such fine task granularity.

$P_\infty = T_1/T_\infty$ is a lower bound on the number of processors necessary to extract the maximum parallelism from the problem. For this problem instance, $P_\infty = 354$ processors. Thus, 354 processors is a lower bound on the number of processors necessary to bring the completion time down to T_∞, namely, 64 seconds. We would like to see what time we could achieve with this many processors, if only our cluster had that many. For the final version of this paper, we will find a problem whose $P_\infty \leq 52$. Then, we can actually push JANET towards the theoretical limit for a specific problem.

CONCLUSIONS AND FUTURE WORK

JANET is a Java-centric distributed service for high-performance parallel computing. We briefly described its API, which includes Cilk-like divide-and-conquer mechanisms plus a globally accessible input object and a best-effort shared object. The architecture has 3 main component types, a hosting service provider which manages a network of task servers, each of which has an associated, dynamic set of compute servers, called hosts. Computations proceed correctly in the presence of host failure. Experimental results show

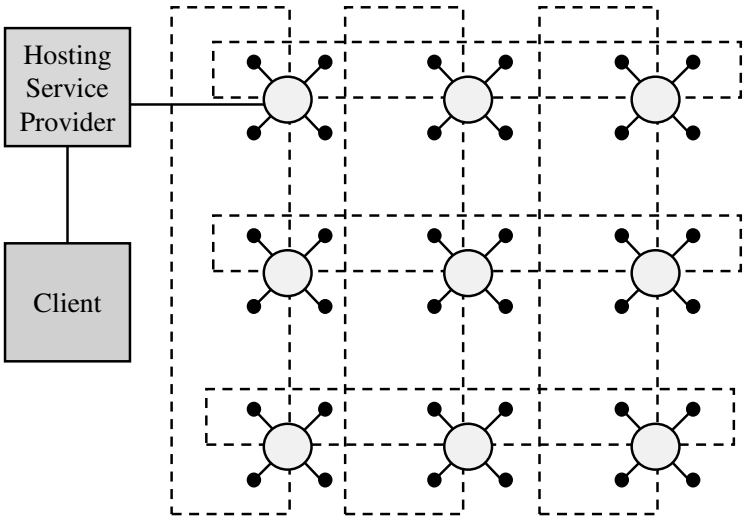

Figure 2: This figure illustrates an instance of the JANET architecture in which there are 9 task servers. The task server topology is a 2D torous (the dashed lines). In the figure, each task server has 4 associated hosts (the little black discs). The client interacts *only* with the HSP.

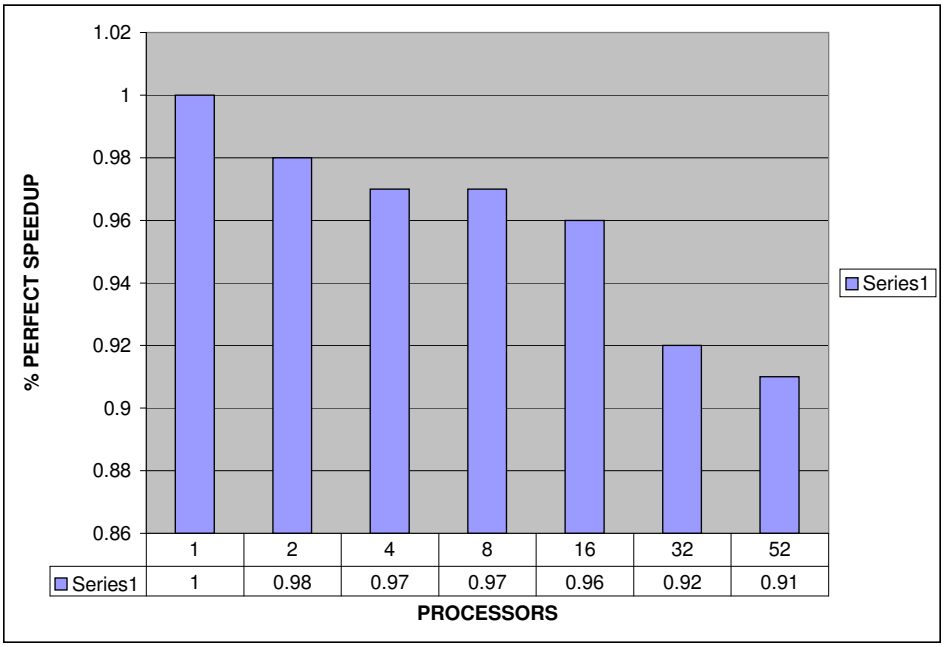

Figure 3: Number of processors vs. % of ideal speedup.

that it maintains excellent speedup: It performed a 150-city traveling salesman problem via branch-and-bound using 1 processor in 6 hours and 18 minutes, while doing the same problem using 52 processors in 8 minutes, attaining at least 91% of ideal speedup.

The API includes a simple set of application-controlled directives for improving performance by reducing communication latency or overlapping it with task execution. We intend to perform more detailed performance experiments to get quantitative data on the effectiveness of the 3 latency-hiding techniques: caching tasks, prefetching tasks, and executing tasks on the task server. We intend to experimentally determine the number of hosts that a task server can serve, as a function of the average task time. This information may move us further in the direction of autonomic computing, dynamically provisioning task servers and hosts according to (moving) average task execution times, for example.

JANET currently gathers task execution times per task class per host per task server. If this information is combined with host characteristics, we could construct a scheduler that is aware of particular host and task characteristics, with the goal of shortening the computation's critical path.

We expect to interface JANET with Globus via the latter's service discovery mechanisms. We also expect to exploit Jini's service discovery mechanisms for self-organization and security. We may attempt to implement JANET on top of Ibis, to see what communication benefits may accrue. Longer term, we would like to put a market layer on top of service discovery, to explore markets in computation [39, 12] between competing hosting service providers.

References

[1] Folding@home. http://folding.stanford.edu/.

[2] GridEngine. http://www.sun.com/grid/.

[3] Parabon Computation. http://www.parabon.com/.

[4] Screensaver Lifesaver. http://www.chem.ox.ac.uk/curecancer.html.

[5] SETI@home. http://setiathome.ssl.berkeley.edu/.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proc. 7th ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[11] 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.

[12] 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.

[13] P. Cappello and D. Mourloukos. A Scalable, Robust Network for Parallel Computing. In *Proc. Joint ACM Java Grande/ISCOPE Conference*, pages 78 – 86, June 2001.

[14] P. Cappello and D. Mourloukos. CX: A Scalable, Robust Network for Parallel Computing. *Scientific Programming*, 10(2):159 – 171, 2001. Ewa Deelman and Carl Kesselman eds.

[15] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of Super Computing*, Nov. 2000. Dallas, TX.

[16] A. Chien. Entropia. http://www.entropia.com/.

[17] B. N. Chun and D. E. Culler. REXEC: A Decentralized, Secure Remote Execution Environment for Clusters. In *Proc. 4th Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Jan. 2000. Toulouse, France.

[18] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[19] T. Fahringer and A. Jugravu. JavaSymphony: New Directives to Control and Synchronize Locality, Parallelism, and Load Balancing for Cluster and GRID-Computing. In *Proc. ACM Java Grande - ISCOPE Conf.*, pages 8 – 17, Nov. 2002.

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

[21] 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.

[22] J. Frey, T. Tannenbaum, I. Foster, M. Livny, , and S. Tuecke. Condor-G: A Computation Management Agent for Multi- Institutional Grids. In *Proc. Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, Aug. 2000. San Francisco, CA.

[23] D. Gelernter and D. Kaminsky. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. In *Proc. Sixth ACM Int. Conf. on Supercomputing*, July 1992.

[24] A. Grimshaw. Avaki. http://www.avaki.com/.

[25] A. S. Grimshaw, W. A. Wulf, and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, Jan. 1997.

[26] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a Framework for Preparing and Executing Adaptive Grid Programs. In *Proc. NSF Next Generation Systems Program Workshop (Int. Parallel and Distributed Processing Symp.)*, Apr. 2002. Ft. Lauderdale, FL.

[27] M. Milenkovic, S. H. Robinson, R. C. Knauerhase, D. Barkai, S. Garg, V. Tewari, T. A. Anderson, and M. Bowman. Toward Internet Distributed Computing. *IEEE Computer*, pages 38–46, May 2003.

[28] M. O. Neary and P. Cappello. Internet-Based TSP Computation with Javelin++. In *1st International Workshop on Scalable Web Services (SWS 2000), International Conference on Parallel Processing*, Toronto, Canada, Aug. 2000.

[29] M. O. Neary and P. Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. In *Proc. ACM Java Grande/ISCOPE Conference*, pages 56 – 65, November 2002.

[30] M. O. Neary and P. Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. *Concurrency and Computation: Practice and Experience*, 2004. To appear.

[31] M. O. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0: Java-Based Parallel Computing on the Internet. In *Euro-Par 2000*, pages 1231–1238, Munich, Germany, Aug. 2000.

[32] T. Sandholm, S. Suri, A. Gilpin, and D. Levine. Winner Determination in Combinatorial Auction Generalizations. In *Proc. of AAMAS: Autonomous Agents and Multiagent Systems*, 2002. Italy.

[33] 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.

[34] R. Seed and T. Sandholm. A Note on Globus Toolket 3 and J2EE. Technical report, Globus, Jan. 2003.

[35] R. Seed, T. Sandholm, and J. Gawor. Globus Toolkit 3 Core — A Grid Service Container Framework. Technical report, Globus, Jan. 2003.

[36] 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.

[37] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Proc. ACM Java Grande - ISCOPE Conf.*, pages 18 – 27, Nov. 2002.

[38] G. von Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke. CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids. In *ACM Java Grande Conference*, June 2000.

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

[40] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. 18th Symp. Operating Systems Principles*, Oct. 2001. Lake Louise, Canada.

[41] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running EveryWare on the Computational Grid. In *Proc. of SC99*, Nov. 1999.