Proceedings of the 17th IASTED International Conference
Parallel and Distributed Computing and Systems
November 14-16, 2005, Phoenix, AZ, USA

JICOS: 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
email: {cappello | ccoakley}@cs.ucsb.edu

ABSTRACT
JICOS is a Java-centric distributed service for high-performance parallel computing. Its API is especially suited to divide-and-conquer computations. Computational tasks can access a global read-only *input* object and a mutable object that is shared asynchronously. These features enable an elegant expression of branch-and-bound optimization, which is used as the benchmark for 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. Experimental results confirm high parallel efficiency on branch-and-bound. Experiments also confirm efficient recovery from host failures. JICOS reports a computation's actual critical path time, which can be used to calculate the maximum available parallelism of a problem instance.

KEY WORDS
cluster, distributed, grid, parallel, divide-and-conquer, branch-and-bound

1 Introduction

For high-performance, distributed computing services, there is increasing interest in supporting *more complex algorithm classes than Master-Worker* by systems: whose compute servers may join and/or leave during program execution; that tolerate faulty compute servers during program execution; and that scale to a large number of computer servers with good speedup. Branch-and-bound constitutes a large class of algorithms. Its implementation is more complex than master-worker for at least 3 reasons: First, the fixed-depth task decomposition tree associated with Master-Worker generalizes to a dynamic-depth task decomposition tree. Second, 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. Thirdly, 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 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 [17, 9, 12, 10, 11].

The contribution of this paper to the study of high-performance, adaptively parallel cluster/network computing is to integrate architectural and language constructs that enable the elegant expression of branch and bound computation, and a distributed implementation that 1) exhibits good speedup, even for small tasks (e.g., 2 sec.), and 2) efficiently tolerates the failure of compute servers.

Background

Java's attraction as a distributed computing platform includes its solution to the portability/interoperability problem associated with heterogeneous machines and OSs. Please see [16] for an enumeration of other advantages. The use of a virtual machine is a significant difference between Java-based systems and previous systems. Java-based efforts are orthogonal to Grid efforts: In principle, they can be loosely coupled to the Grid via Grid protocols. The Java CoG Kit[16] facilitates such couplings. 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. We focus on systems in category 2, which can be further subdivided into: 1) those that support adaptive parallelism (originally defined in [7]); 2) those that do not. We focus on the first category, which includes Popcorn [4], Charlotte [2], Atlas [1], Javelin, Bayanihan [13], and Satin [18], among others. Like Atlas, CX [5, 6] supports a divide-and-conquer API, and adaptively parallel algorithms with faulty processors. Satin is most like Jicos: It is Java-centric, tolerates faulty compute servers, and supports small-grained, adaptively parallel computation with a

divide-and-conquer API. It is compatible with large-scale cluster settings, LAN/WAN, and corporate intranets. However, unlike Satin, all tasks in Jicos have access to a common environment consisting of an immutable *input* object and a mutable object that is *shared* asynchronously.

2 API

Computation is modelled by an *directed acyclic graph* (DAG) 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. Tasks have access to a *shared object*. The semantics of "shared" reflects the envisioned computing context—a computer network: The object is shared *asynchronously*. 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.

Divide and conquer

Tasks are central to the API; they encapsulate computation via the execute method:

```
public Object execute(Environment environ);
```

The Environment object is described below. Let A be a task. It has an array of input objects (possibly of length 0). It returns either an output Object, which is an input to its successor task, A', or it decomposes into a sequence of subtasks and returns a *compose* task. Each subtask produces an output that is an input to the compose task. The return value of the compose task is the output associated with task A, an input of task A'. Thus, the decomposition of a task into subtasks occurs in one task; the composition of the subtasks into a solution occurs in another task. Separating task decomposition from subtask composition is called *explicit continuation passing* [3]. It improves host utilization and enables *efficient recovery from host failure*. From the foregoing, we see that the DAG of tasks is revealed to Jicos only at execution time: Task scheduling is done at execution-time.

The common environment

A computation's tasks access a common Environment object, which has the read-only input object for the entire computation (not an input for a particular task) and the mutable shared object. 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 mutate it via the setShared method. A shared object implements the `Shared` interface, which includes the *isNewerThan* method:

```
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 asynchronously. 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 itself, via its 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 locally best known tour. In a distributed setting this may not always be the case, hence the need to check.

3 Architecture

JICOS, 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 [14, 15] frontend for service discovery (not presented), is largely independent of hardware/OS, and is intended to scale from a LAN to the Internet. JICOS 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**: JICOS 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. JICOS comprises 3 service component classes.

Hosting Service Provider (HSP): JICOS 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 network of task servers described below. For example, when a task server wants to join the distributed service, it first contacts the hosting service provider. The HSP tells the task server where it fits in the task server network.

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 1.

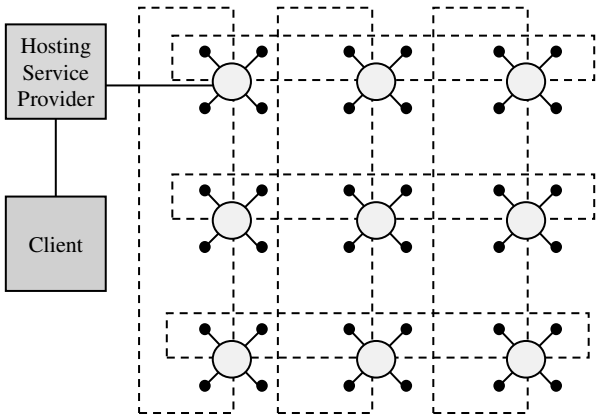

Figure 1. A JICOS system that has 9 task servers. The task server topology, a 2D torous, is indicated by the dashed lines. In the figure, each task server has 4 associated hosts (the little black discs). The client interacts *only* with the HSP.

Task objects *encapsulate* computation: Their inputs & outputs are managed by JICOS. Task execution is idempotent, supporting the requirement for host transience and failure recovery.

HIDING COMMUNICATION LATENCY

JICOS'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 its host, obviating its host's need to ask the TaskServer for its next task. *The application programmer thus implicitly controls which subtask is cached.*

Task pre-fetching The *application* can help hide communication latency via task pre-fetching:

Implicit: A task that never constructs subtasks is called *atomic*. The Task class has a boolean method, *isAtomic*. The default implementation of this method returns true, if and only if the task's class implements the marking interface, *Atomic*. Before invoking a task's *execute* method, a host invokes the task's *isAtomic* method. If it returns true, the host pre-fetches another task via another thread before invoking the task's execute method.

Explicit: When a task object whose *isAtomic* method returned false (it did not know prior to the invocation of its *execute* method that it would not generate subtasks) nonetheless comes to a point in its *execute* method when knows that it is not going to construct any subtasks, it can invoke its environment's *pre-fetch* method. This causes its host to request a task from the task server in a separate thread.

Task pre-fetching overlaps the host's execution of the current task with its request for the next task. Application-directed pre-fetching, both implicit and explicit, thus motivates the programmer to 1) identify atomic task classes, and 2) constitute atomic tasks with compute time that is at least as long as a Host—TaskServer round trip (on the order of tens 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, it is faster for the task server to execute the task itself than to send it to a host for execution. 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). Binary boolean operators, such as min, max, sum (typical linear-time gather operations) should execute on the task server. All Task classes have a boolean method, executeOnServer. The default implementation returns true, if and only if the task's class implements the marking interface, *ExecuteOnServer*. When a task is ready for execution, the task server invokes its executeOnServer method. If it returns true, the task server executes the task itself: *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.

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. Because of explicit continuation passing, recomputation is minimized: Systems that support divide-and-conquer but which do not use explicit continuation passing, such as Satin [18], need to recompute some task decomposition computations, even if they completed successfully. In some applications, such as TSP, decomposition is computationally complex. On Jicos, only the task that was currently being executed needs to be recomputed. This is an architecturally significant improvement. In the TSP instance that we use for our performance experiments, the average task time is 2 sec. Thus, the recomputation time for a failed host is, in this instance, a mere 1 sec, on average.

4 Performance Experiments

THE TEST ENVIRONMENT

We ran our experiments on a Linux cluster. The cluster consists of 1 head machine, and 64 compute machines, composed of two processor types. Each machine is a dual 2.6GHz (or 3.0GHz) Xeon processor with 3GB (2GB) of PC2100 memory, two 36GB (32GB) 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 machine is running CentOS 4 with the Linux smp kernel 2.6.9-5.0.3.ELsmp, and the Java j2sdk1.4.2. Hyperthreading is enabled on most, but not all, machines.

THE TEST PROBLEM

We ran a branch-and-bound TSP application, using kroB200 from TSPLIB, a 200 city euclidean instance. In an attempt to ensure that the speedup could not be superlinear, 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 61,295 BranchAndBound tasks whose average execution time was 2.05 seconds, and exactly 30,647 MinSolution tasks whose average execution time was less than 1 millisecond.

THE MEASUREMENT PROCESS

For each experiment, a hosting service provider was launched, followed by a single task server on the same machine. When additional task servers were used, they were started on dedicated machines. Each host was started on its own machine. Except for 28 hosts in the 120 processor case (which were calibrated with a separate base case), each host thus had access to 2 hyperthreaded processors which are presented to the JVM as 4 processors (we report physical CPUs in our results). After the JICOS system was configured, a client was started on the same machine 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 Jicos'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. Jicos also automatically computes the critical path using the obvious recursive formulation for a DAG. Each test was run 8 times (or more) and averages are reported.

Due to hyperthreading, 1 processor in the OS does not correspond to 1 physical processor. It therefore is difficult to get meaningful results for 1 processor. We consequently use 1 machine, which is 2 physical CPUs, as our base case. For the 120 processor measurements, we used the heterogeneous speedup formula from CX [6]. We had 3 separate base cases for computing the 120 processor speedup.

For our fault tolerance test, we launched a JICOS system with 32 processors as compute servers. We issued a kill command to various compute servers after 1,500 seconds, approximately 3/4 through the computation. The completion time for the total computation was recorded, and was compared to the ideal completion time: $1500 + (T_{32} - 1500) \times 32/P_{final}$), where P_{final} denotes the number of compute servers that did *not* fail.

To test the overhead of running a task server on the same machine as a compute server, we ran a 22 processor experiment both with a dedicated task server and with a task server running on the same machine as one of the compute servers. We recorded the completion times and report the averages of 8 runs.

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 37 seconds. A well known *lower* bound on the completion time [3] 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 2 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 0.94; it exceeds 0.96, when using an appropriate number of task servers. Specifically, the 2-processor base case ran in 9 hours and 33 minutes; whereas the 120-processor experiment (2 processors per host) ran in under 12 minutes!

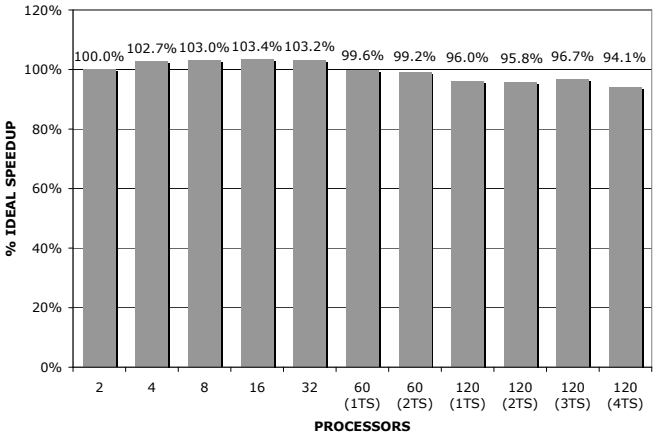

Figure 2. Number of processors vs. % of ideal speedup.

We get superlinear speedups for 4, 8, 16, and 32 processors. The standard deviation was less than 1.6% of the size of the average. As such, the superlinearity cannot be explained by statistical error. However, differences in object placement in the memory hierarchy can have impacts greater than the gap in speedup we observe [8]. So within experimental factors beyond our control, Jicos performs well.

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 2 second tasks. Even CX [5, 6] 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, $P_\infty = 1,857$ processors. Thus, 1,857 processors is a lower bound on the number of processors necessary to bring the completion time down to T_∞, namely, 37 seconds.

Table 1. Fault Tolerance (Times are in seconds)

Processors (Final)	Theoretical Time	Measured Time	Percent Overhead
30	2119.43	2194.95	3.6%
26	2214.73	2300.92	3.9%
12	3048.58	2974.35	-2.4%
8	3822.87	4182.62	9.4%
6	4597.16	4884.86	6.3%
4	6145.74	6559.91	6.7%

Our fault tolerance data is summarized in Table 1. Overhead is caused by the rescheduling of tasks lost when a Host failed as well as some time taken by the TaskServer to recognize a faulty Host. Negative overhead is a consequence of network traffic and thread scheduling preventing a timely transfer of the kill command to the appropriate Host.

When measuring the overhead of running a task server on a machine shared with a compute server, we received an average of 3115.1 seconds for a dedicated task server and 3114.8 seconds for the shared case. Both of these represent 99.7% ideal speedup. This is not too surprising: there is a slight reduction in communication latency having the task server on the same machine as a compute server, and the computational load of the task server is small due to the simplicity of the compose task (it is a comparison of two upper bounds). It therefore appears beneficial to place a compute server on every available computer in a JICOS system without dedicating machines to task servers.

5 Conclusion and Future Work

JICOS 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 an asynchronously shared object. The architecture has 3 main component types, a hosting service provider which mediates clients with a network of task servers, each of which has an associated, dynamic set of compute servers, called hosts. Experimental results show that it maintains excellent speedup: It performed a 200-city TSPLIB traveling salesman problem via branch-and-bound using 1 host (2 CPUs) in 9 hours and 33 minutes, while doing the same problem using 120 CPUs in under 12 minutes, attaining 96.66% of ideal speedup in a heterogeneous environment.

Load balancing the tasks generated by branch-and-bound computations is notoriously difficult. Our speedups are quite good, due in no small part to the Jicos API, which includes a simple set of application-controlled directives for improving performance by reducing communication latency or overlapping it with task execution. Transactions are avoided. We intend to perform more detailed performance experiments to get quantitative data on the effectiveness of the 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.

JICOS 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 time, which is automatically calculated by the system for each client computation.

The fault tolerance of our compute servers is efficient,

both when faults occur (as indicated by our fault tolerance performance experiments), and when they do not (as indicated by our speedup experiments, in which no faults occur). This efficiency is due primarily to JICOS's use of a space-based architecture and explicit continuation passing, which minimizes the need for recomputation. We have a plan, based on the Satin scheme, for enabling Jicos to recover from task server failures.

Finally, the overhead of task servers is shown to be quite small, further confirming the efficiency of JICOS as a distributed system.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] C. Krintz and T. Sherwood. Private communication, 2005.

[9] 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*, pages 137 – 144, Toronto, Canada, Aug. 2000.

[10] 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.

[11] M. O. Neary and P. Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. *Concurrency and Computation: Practice and Experience*, 17:797 – 819, 2005. Published online in Wiley Interscience (www.interscience.wiley.com). DOI: 10.1002/cpe.855. Received 15 Jan 03. Revised 31 Aug 03. Accepted 14 Oct 03.

[12] 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.

[13] 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.

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

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

[16] 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*, pages 97 – 106, June 2000.

[17] 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.

[18] G. Wrzesinska, R. V. van Nieuwpoort, J. Maasen, T. Kielmann, and H. E. Bal. Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. *Int. J. High Performance Computing Applications*. Accepted for publication.