Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing

Michael O. Neary
neary@cs.ucsb.edu

Peter Cappello
cappello@cs.ucsb.edu

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

ABSTRACT

Javelin 3 is a software system for developing large-scale, fault tolerant, adaptively parallel applications. When all or part of their application can be cast as a master-worker or branch-and-bound computation, Javelin 3 frees application developers from concerns about inter-processor communication and fault tolerance among networked hosts, allowing them to focus on the underlying application. The paper describes a fault tolerant task scheduler and its performance analysis. The task scheduler integrates work stealing with an advanced form of eager scheduling. It enables dynamic task decomposition, which improves host load-balancing in the presence of tasks whose non-uniform computational load is evident only at execution time. Speedup measurements are presented of actual performance on up to 1,000 hosts. We analyze the expected performance degradation due to unresponsive hosts, and measure actual performance degradation due to unresponsive hosts.

Categories and Subject Descriptors

D.1.3 [Software]: Programming Techniques—concurrent programming

General Terms

Design, Experimentation, Performance, Reliability

Keywords

Branch-and-bound, eager scheduling, fault tolerance, Grid computing, parallel computing

1. INTRODUCTION

*This work was supported by an NPACI grant.

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: 1) 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); 2) Worker tasks do not communicate among themselves. They merely report their result to the master process. Developing efficient distributed computations for such problems is desirable for at least two reasons: First, such algorithms can be found for many important problems (e.g., protein folding, genetic sequencing, parameter sweep optimization problems). Second, such algorithms have a natural distributed implementation. The number of such distributed systems is growing rapidly (e.g., SETI@home from UC Berkeley, the fight against cancer from Parabon Computation, folding@home from Stanford, the anti-cancer drug discovery project, screensaver lifesaver, from Oxford in collaboration with United Devices, and fight AIDS at home from Entropia).

It is nonetheless important to discover more complex algorithm classes that can be efficiently supported as adaptively parallel computations. Branch and bound is a class that encompasses a large set of algorithms. The simple Master-Worker algorithms, described above, form a degenerate case. Branch-and-bound's implementation is more complex for several 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 not balanced. It can be quite irregular, due to the bounding process (also known as pruning), in a way that depends on the problem instance, and thus is known only at execution time. Finally, tree pruning, to proceed efficiently, requires communication among the compute hosts, as they discover new bounds.

Although Branch-and-bound is challenging to implement efficiently in an adaptively parallel setting, it is a challenge well worth accepting. A wealth of important optimization problems are routinely solved via a branch-and-bound algorithm (e.g., many computationally hard problems such as Integer Linear Programming and combinatorial auction winner selection). Thus, if we can efficiently speed up branch-and-bound in an adaptively parallel setting, then the usefulness of, and demand for, adaptively parallel computing will increase dramatically.

It is difficult to implement adaptively parallel branch-and-bound so that speedup scales to large processor sets. Our
The speedup scalability problem concerns the decision as to which tasks should be further decomposed, and which should not (and thus be completely solved within a single compute host). To understand this scalability problem, we must understand the branch-and-bound process. A branch-and-bound algorithm seeks an optimum solution from among a set of feasible solutions. The size of this set is typically exponential in the size of the original problem. For example, the set of feasible tours in a Traveling Salesman Problem (TSP) of a complete graph with 23 nodes is \( 22! \) or around \( 1.6 \times 10^{15} \) tours. The space of feasible solutions is progressively partitioned (branching), forming a problem tree. Each node contains a partial feasible solution. The node represents the set of feasible solutions that are extensions of its partial solution. As branching continues (progresses down the problem tree), the nodes contain more complete partial solutions, and thus represent smaller sets of feasible solutions. For example, when casting the TSP as a branch-and-bound problem, a node in the problem tree contains a partial tour, and represents the set of all tours containing that partial tour. As one progresses down the nodes of the problem tree, the nodes represent larger partial tours. As the size of a partial tour increases, the number of full tours containing the partial tour clearly decreases.

Tasks correspond to nodes in the problem tree: Each gives rise to a set of smaller tasks until the task represents a node in the tree that is small enough to be explored by a single host. Such tasks are called atomic. One can programatically define atomic tasks as those whose corresponding problem sub-trees are of a fixed height, \( k \), possibly a function of the problem tree’s height (see Figure 3). In the TSP case, such an atomic task represents no more than \( k! \) feasible tours. For example, in an 18-node input graph, if \( k = 11 \), there is \( 17! / 11! \) or about 9 million atomic tasks, each of which represents 11! or about 40 million tours! Although an atomic task is computed on a single host, the number of tours that actually need to be explored depends on how much of the task’s corresponding sub-tree can be pruned, which depends on the problem instance. Consequently, the number of tours to be examined, in this example, can vary from 0 to about 40 million! Having atomic tasks whose computational load is so variable makes high speedups problematic, when deploying on a large number of processors.

We cannot have \( k \) be a function of the number of processors; in adaptively parallel computing, this number varies at execution time. Work-stealing is distributed among the compute hosts: Its parameters cannot include global properties of the distributed system. Neither can we reduce the maximum variation in the atomic task’s computation load by simply reducing \( k \); this substantially increases the number of tasks. The overhead of managing a large number of tasks becomes prohibitive. For example, reducing \( k \) to 5 in an 18-node graph increases the number of tasks from 40 million to almost 14 billion! We thus are motivated to vary the atomic threshold at execution time.

This paper presents a novel integrated work-stealing, eager scheduler that efficiently schedules and reschedules tasks, dynamically varying the atomic threshold according to the perceived load characteristics of the adaptively parallel computation (failed compute hosts are indistinguishable from hosts that respond very slowly either because they are very slow, or because they are working on a very large atomic task, or some combination thereof). Experimental results show good speedup in our benchmark branch-and-bound problem, TSP, using as many as 1,000 compute hosts. We expect comparable speedups on larger numbers of hosts for larger problem instances. Since the scheduler balances the load among compute hosts and reschedules tasks that are assigned to failed hosts, we demonstrate an implementation of adaptively parallel branch-and-bound that both scales to a large number of compute hosts and tolerates faults (both node and link).

The system is pure Java for reasons that include supporting heterogeneous resources. Measuring speedup is a focus of this paper, however. This requires a controlled experimental environment of dedicated processors. For our large speedup experiments, we used the San Diego Supercomputing Center’s IBM Blue Horizon, which happens to be a homogeneous multiprocessor. Our earlier experience with heterogeneous environments, albeit smaller ones, has been reported elsewhere [16].

The principle contributions of this paper to the study of adaptively parallel computing are:

- the novel dynamic depth eager scheduling method, and
- an analysis, and experimental validation, of the quantity of degradation due to unresponsive hosts.

### 1.1 Background

A few years ago a pair of developments ushered in a new era. One was the announcement of the SETI@home project. Since then, similar projects have arisen (see above). While such applications are of the simple Master-Worker variety, they promote a vision of a world-wide virtual supercomputer [11, 1] or globally distributed computations.

The related development was an emerging vision of The Grid, an “integrated, collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations.” [10]. The Globus project [10] is the best known manifestation of that vision. In contrast to Globus, which is open source, Avaki is a proprietary version of that vision, rooted in Legion research [14]. Such systems are not Java-centric, and indeed must be language-neutral. The Millennium Project [8] is a “system of systems” project where a key element of the research is how constituent systems interact. The Ninja project [23] 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. [6], and Condor-G by Frey et al. [12] (where Condor [9] makes use of several grid services, such as GSIFTP). EveryWare [24] 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”[24]. 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 distributed heterogeneous applications destined for the Grid [15].

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 [22] for an enumeration of other advantages. 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[22] 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 is determined 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. Manta [21], another example, elegantly shows that wide-area networks, modeled as two-level networks, can efficiently support large, coarse-grain parallel computation. Systems of category 2 can be further subdivided: A) those that support adaptive parallelism (originally defined in [13]); and B) those that do not. Category B) 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). Category A) includes Charlotte [3], Atlas [2], Javelin, and Bayanihan [20], among others. 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 [5] 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. Nihhanupudi et al. [19] present work on adaptive BSP, an efficient, programmer-friendly model of parallel computation suitable for the harvesting of idle cycles in a LAN.

Javelin is Java-based, and supports coarse-grained, adaptively parallel computation. It is compatible with large scale cluster settings, LAN/WAN, and even corporate intranets.

2. ARCHITECTURE

This section briefly introduces the basic architectural concepts of Javelin 3. It is intended for readers who are not familiar with Javelin; for more details, see [7, 16, 18].

The Javelin 3 architecture retains the basic structure of its predecessors, Javelin and Javelin++. 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. Each process maps to a single Java Virtual Machine. Figure 1 illustrates the architecture. Clients register their tasks to be run with a broker; hosts register their intention to run tasks with a broker. The broker assigns tasks to hosts that, then, run the tasks and send results back to the clients. The role of a host or a client is not fixed. A machine may serve as a Javelin host when it is idle, while being a client when its owner wants additional computing resources.

![Figure 1: The Javelin 3 Architecture.](image)

2.1 Broker Network & Host Tree

The topology of the broker network is an unrestricted graph of bounded degree. Thus, at any time a broker can only communicate with a constant number of other brokers. This constant may vary among brokers according to their communicational/computational power. Similarly, a broker can only handle a constant number of hosts. If that limit is exceeded, hosts must be redirected to other brokers. The bounds on both types of connection give the broker network the potential to scale to arbitrary numbers of participants. Figure 2 shows the connection setup of a broker.

![Figure 2: Broker Connections.](image)

When a host connects to a broker, the broker enters the host in a logical tree structure. The top-level host in the tree does not receive a parent; instead it later becomes a child of the client. This way, the broker maintains a pre-organized 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 with a heap-like operation (see Section 3.3).
2.2 The Work Stealing Scheduler

Task stealing in Javelin 3 is done via work stealing, a distributed scheduling scheme made popular by the Cilk project [4]. 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 per host is high — a property well suited for adaptively parallel systems.

In Javelin 3, 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.

2.3 Shared Memory for Bound Propagation

In the master-worker model of computation, shared memory is not needed. It might appear that we cannot implement shared memory efficiently among networked processors in a manner that scales; the communication latency is too large. However, for branch-and-bound computation:

- Only a small amount of shared memory is needed, because only one int or double is needed to represent a solution’s cost.
- A weak shared memory model suffices; 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.

In Javelin 3, 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 propagate it to their children, and so on, down the host tree.

3. ADVANCED EAGER SCHEDULING

Eager scheduling reschedules a task to an idle processor, if in case its result has not been reported. It was introduced and made popular by the Charlotte project [3]. It efficiently and relentlessly progresses towards the overall solution in the presence of host and link failures, and varying host processing speeds. In addition, it also balances the computational load. The Javelin 3 eager scheduler is located in the client process, which is at the root of the host tree. Although this may seem like a bottleneck with respect to scalability, it is not, as we shall explain below.

3.1 The Master-Worker Eager Scheduler

The master-worker eager scheduler is unchanged from earlier versions of the system, Javelin++ [16] and Javelin 2 [18]. It uses a fixed-depth, tree-based scheme with a circularly linked list of undone tasks.

3.2 The Branch-and-Bound Eager Scheduler

Eager scheduling is more challenging for branch and bound computation. To begin with, the computation produces two types of result, which must be handled differently:

1. A positive result is a new best cost solution. It is propagated to all hosts as soon as possible, and leads to more efficient pruning of the search tree.
2. A negative result is a solution subspace that has either been fully examined, or pruned from the search tree.

In comparison, a master-worker computation only produces “negative” results under this terminology. The eager scheduler does not handle positive results in Javelin 3; we use the bound propagation mechanism described in Section 2.3. Negative results are handled by the eager scheduler.

In a branch-and-bound computation, the size of the feasible solution space is typically exponential in the size of the input. The algorithm may need to examine all feasible solutions to find the minimum cost solution. A partial solution, $p$, is “pruned” when the lower bound on the cost of any feasible solution that is an extension of $p$ is more costly than the currently known minimum cost solution. To detect problem termination, the eager scheduler eventually must know that a node is pruned or represents only sub-optimal solutions. If a separate communication is required to inform the eager scheduler of each such node, the quantity of communication would nullify the benefits of parallelism. Three mechanisms are used to avoid communication overload. First, whole subtrees of the search space are packaged as a single, atomic task. Second, information about pruned nodes is computed by the eager scheduler lazily. The scheduler must know what nodes can be pruned when no work is left to steal. However, at that time, it may have a smaller upper bound, enabling it to prune more of the problem tree. Thirdly, hosts aggregate negative results over a time interval (e.g., 5 seconds), sending them to their parent only periodically. To reduce runtime, we adjust the computation/communication ratio by adjusting the initial size of atomic tasks.

Figure 3 shows an example of the search tree division into atomic and non-atomic tasks. Here, the atomic depth parameter is set to 2, which means all tasks that are 2 nodes below the root node are considered atomic and are processed by hosts with no further subdivision. Many of these tasks may in fact be pruned away, due to an existing better minimum cost bound. We have highlighted two portions of the tree that show the extent of local computation of an atomic task. The eager scheduler itself never processes any portions of the search tree below the atomic depth level, although it will analyze the top portion of the tree based on negative results and its own min cost bound.

3.2.1 Deferred Detection of Pruned Nodes

The branch-and-bound eager scheduler employs lazy or deferred detection of pruned nodes, which takes place only after the initial round of work stealing has ended, and the first host finds itself out of work and sends a work request to the eager scheduler. At this point the eager scheduler infers
A to m ic d e p th = 2
A to m ic ta s k

Figure 3: Atomic Tasks and Atomic Depth.

what nodes representing partial solutions have been pruned, and which atomic tasks need to be rescheduled.

3.2.2 Method

The basic data structure required is a problem tree, which the eager scheduler, located in the client process, maintains to keep track of the computation status. Each atomic task is a leaf in this problem tree. The root of the problem tree represents the complete branch-and-bound computation. Its children are the subproblems resulting from branching — a single split of the root problem. This branching (splitting) continues, as we proceed down the problem tree: We subdivide it into smaller and smaller search spaces. A parameter, the atomic depth, determines at what level splitting stops. At that point, a host will search the space for a solution that is less than the current minimum cost solution.

Each node (task) in the problem tree can be in one of 3 states: done, meaning the results for the subproblem have been received by the eager scheduler; partially done, meaning that results have been received by the eager scheduler for some but not all descendants of this subproblem (i.e., some but not all subproblems of this subproblem); and undone, meaning that no results have been received by the eager scheduler for this subproblem. In addition to the tree structure, undone tasks are put in a circular linked list. Tasks are eagerly scheduled from this circular list until it becomes empty: there are no undone tasks. This indicates completion of the computation; the eager scheduler then propagates a termination signal down the host tree. The processing itself consists of two distinct routines: result processing and task selection, given below.

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

public Task selectTask() {
    // esTask refers to last eagerly scheduled node
    while ( (esTask = esTask.next() ) != null ) {
        generate esTask’s children & their costs;
        insert children into ProblemTree;
        maintain undone task list;
        esTask = selectTask( 1 of these children );
    }
    if ( esTask.isAtomic() ) {
        if ( esTask.hasBeenRescheduled() ) {
            // dynamic depth increment
            esTask.incrementAtomicDepth();
            continue;
        }
        return esTask; // have atomic node to process
    }
    return null;
}
```

Figure 4 illustrates Javelin 3 eager scheduling. In Figure 4(a), we see how the results of the atomic tasks with IDs 4 and 6 have arrived at the eager scheduler. The eager scheduler subsequently marks these tasks as done, and all their ancestors including the root as partly done. Figure 4(b) shows how, assuming no further results arrive at the eager scheduler and work stealing has failed, the eager scheduler analyzes the current situation: based on its current min cost bound, it infers that the task with ID 3 was pruned from the tree, and subsequently marks task 1 as done. Finally, in Figure 4(c), we show how the eager scheduler selects task 5 as the next undone piece of work. This task will now be reissued for host computation.

```java
Figure 4: (a) Results 4 & 6 arrive. (b)Deferred detection of pruned task 3. (c) Task 5 selected.
```

3.2.3 Dynamic Depth Expansion

Our experience has shown that, due to the inherent irregularity of many branch-and-bound problems (e.g., the TSP), computation times for tasks with a fixed atomic depth parameter vary greatly; in fact, some pieces compute in milliseconds, as they are quickly pruned; others may take more than half the time of the complete computation. Figure 5(a) shows a situation in which a single large atomic task forms the bulk of the computation.

We therefore improved the fixed-depth eager scheduling scheme described in [18] by adding a dynamic depth component, which increases the atomic depth of eagerly scheduled pieces on each new round. Thus, larger tasks can be sub-divided into tasks representing smaller subproblems and distributed to several hosts. This dynamic depth expansion has greatly improved the performance and scalability of our system (see Section 4). Figure 5(b) shows an example of
this mechanism: a large atomic task is sub-divided by incrementing the atomic depth. The process can be repeated, as shown in Fig. 5(c), if necessary.

### 3.3 Repairing the Host Tree

If a host located at a leaf position in the tree fails (or retreats from the system), it does not affect other hosts very much, since the failed host, when detected, is just taken off the detecting host’s address list. In case of a non-leaf host failure, however, a host failure has more consequences. Depending on its position in the host tree, a failed host blocks communication among its children, and a portion of the undone computation residing at the failed host may never be assigned to any host in its subtree. Eager scheduling guarantees that the work is done eventually by the hosts that remain accessible to the client, but it clearly is desirable to fix a broken tree structure as fast as possible, especially if the failed host is the root of a large subtree of hosts. Javelin 3 automatically fixes a tree as soon as a host failure is detected. As a precondition, we assume that the broker is a *stable participant*, since it runs the tree manager.

The tree repair scheme works as follows: When a host is assigned a position in the tree, it is given information on how to contact its parent. If a host later detects that its parent is dead, it immediately notifies the broker of this condition. If the empty position has already been reported and filled, the tree manager traverses the tree representation, and returns the new parent to the host. However, if the host is the first to report the failure, the tree manager re-heaps the tree. Figure 6 illustrates the situation where node 1 has failed and is replaced by node 6, which is moved to its new position.

![Figure 6: Host 1 Fails, Host 3 Detects Failure; Broker Preempts Host 6.](image)

At present, the tree repair scheme only can cope with host failures, i.e., all hosts that detect a failure must agree on that observation. A single host is not able to distinguish between host and link (communication) failures; the result is the same from the point of view of the host. In case of a link failure between a host and only one of its children, the present scheme reports a failure to the broker even when sibling hosts can still communicate with the “failed” parent host. Clearly, what is needed is a form of *quorum consensus* algorithm. Therefore, our basic scheme needs to respond in a more sophisticated way to link failures. This is a topic of future research in Javelin 3.

### 3.4 Eager Scheduling Analysis

#### 3.4.1 Leaf Node Failures

Let $T_n$ denote the running time of an experiment with $n$ hosts with identical compute power. If a leaf node fails in the host tree, eager scheduling guarantees that all lost work, i.e., all the pieces that the failed host was currently working on or that had not been reported back to the eager scheduler yet, eventually is rescheduled. Thus, an trivial upper bound on the running time with a single failure is $T_{n-1}$. By the same reasoning, if there are $k$ leaf node failures, the expected running time is at least as fast as $T_{n-k}$. We assume the client to be a *stable participant*; thus, in the worst case (all other hosts fail), the expected running time degenerates to $T_l$. If we make a few more assumptions, we obtain a much tighter bound on the expected running time:

1. The time to calculate an individual, atomic piece of work $\Delta t$ is negligible in comparison to the total compute time of the problem $T$, or $\Delta t \ll T$.
2. The average communication latency $l$ between hosts is negligible in comparison to $\Delta t$, or $l \ll \Delta t$.
3. The total number of failures $k$ is small compared to the number of initial hosts, $n$.

In the absence of failures, with $n$ identical hosts participating in the computation, each host contributes approximately $1/n$ to the computation. Under the above assumptions, if a failure occurs at time $t$ somewhere between the beginning of the computation, $t_0$, and the end, $t_1$ (with $T = t_1 - t_0$), then the contribution of the failed host is approximated by $\Delta t/n$. A special case occurs when the failure occurs exactly halfway through the computation, as shown in Figure 7(a). Here, the contribution of the failed host is $1/2n$. This means that, if 2 hosts fail mid-way through the computation, the effect is as if we had one fewer host throughout the whole computation. Let $E_T(t)$ denote the expected running time for $i$ failures. Hence, $E_T(2) = T_{n-1}$, or, for any $k$ failures, the expected running time

$E_T(k) = T_{n-k}/2$.

We now show that the above result holds even under failures that don’t occur exactly halfway through the computation: For instance, if two failures occur at times that are symmetric to the halfway point, the effect is as if both failures had occurred mid-way. Figure 7(b) illustrates the example.

Consequently, even under the assumption of a *constant failure rate*, as depicted in Figure 7(c), the effects of failures combine pairwise, and the above formula holds — as long as failures are approximately symmetric to the halfway point. Obviously, if the failure rate is too high, the boundary assumptions are violated, and performance degenerates.
3.4.2 Non-Leaf Node Failures & Tree Repair

We now look at what happens if a non-leaf node, i.e., a host situated higher up in the tree, fails. Depending on its position in the host tree, a failed host blocks communication among its children, and a portion of the undone computation residing at the failed host may never be assigned to any host in its subtree. Eager scheduling guarantees that the work is done eventually by the hosts that remain accessible.

Let \( h(i) \) denote the height of node \( i \) in the host tree, with the height of a leaf node being 0. Let \( b \) denote the branching factor of the tree. Then, a failed node cuts off \( k = \sum_{i=0}^{b(i)} b^i \) hosts in the tree. Since the cut off hosts are treated like any other failed hosts by the system, we simply plug this number into the above formula to predict the overall running time.

Under the tree repair scheme described above, the consequences of a host failure higher up the tree are again reduced to a leaf node failure, and the above formula holds.

4. EXPERIMENTAL RESULTS

We begin with a series of experiments designed to demonstrate the scalability of our system, followed by experiments to validate the preceding fault tolerance performance analysis. For these tests, we had access to three dedicated parallel machines:

1. A 96-processor Beowulf cluster at our Computer Science Department at UC Santa Barbara. This machine has six 500 MHz Pentium III quad-processor nodes, and 36 400 MHz Pentium II dual processors, each with 512 MB or 1 GB of memory, running Red Hat Linux 6.2.

2. A 192-processor Beowulf cluster at University of Paderborn, Germany. This machine has 96 850 MHz Pentium III dual-processor nodes, each with 512 MB memory, running Red Hat Linux 7.1.

3. An 1152-processor IBM Blue Horizon at NPACI, San Diego\(^1\). The machine has 144 SP Power3 8-processor nodes, clocked at 375 MHz, with 4 GB of memory per node. The OS is AIX.

\(^1\)According to the latest Top 500 list, this is currently the 18th most powerful machine in the world.

We tested our system on these dedicated architectures in order to obtain clean speedup curves in the classical sense, and to work in a controlled test environment. Previously, we published results obtained on a LAN [16].

4.1 Scalability Experiments

For the scalability experiments, we chose to run a classical branch-and-bound application, the Traveling Salesman Problem (TSP). In brief, the TSP can be stated as follows:

Given a weighted, directed graph \( G = (V, E) \), find a minimum weight tour such that each \( v \in V \) is visited exactly once.

Figure 8 shows a simple instance of the TSP. Here, the graph is undirected, which can be viewed as a special case of a directed graph with both corresponding directed edges having identical weight. The minimum tour for this instance is highlighted in the figure.

Figure 8: An Instance of the TSP.

Figure 9 shows the corresponding complete search tree for this simple instance. From this it becomes clear that even for very small instances of the problem, search trees can be huge. In fact, it is well known that the TSP falls into the category of NP-complete problems, for which all currently known algorithms leading to a general solution have at least exponential complexity.

Figure 9: A TSP Search Tree.

Our TSP application is a simple, depth-first algorithm for local computation. A breadth-first component is introduced by running the problem in parallel. The current version is optimized (compared to the results published in [18]) for local computation via a Kruskal Minimum Spanning Tree lower bound. While this led to dramatic improvements in local computation speed, and let us process much larger graphs (35–37 nodes, instead of 22–24 nodes previously), the task of achieving good speedup and scalability is harder, since communication latencies have not improved.
4.1.1 Finding Suitable Input Graphs

For our largest experiments, the test graphs were complete, undirected, weighted graphs of 35-37 nodes, with randomly generated integer edge weights \( w, 0 \leq w < 1000 \). These graphs were complex enough to justify parallel computing, but small enough to enable us to run tests in a reasonable amount of time. By using complete graphs as input, we made sure that the graphs were as dense as possible, making this type of graph the hardest to process for a given size. Also, we are not exploiting any special cases like for instance in the Euclidian TSP, where all edge weights fulfill the triangle inequalities, and which can therefore be bounded from above by \( 2 \times \) (the weight of its minimum spanning tree).

The procedure of finding input graphs itself is tedious: we generated some 45 candidate graphs over time, and tested them on a small set of hosts — initially 20, later 64 — to get an idea of the problem size. Many of the candidate graphs proved either too simple (running time too short) or too hard (running time too long). We considered graphs that ran in approximately 10-20 hours on 64 processors of the Paderborn Linux cluster. Our aim was to find graphs that would run for 2-4 hours on 1000 processors on NPACI’s IBM Blue Horizon. Table 1 shows some characteristic data for the graphs that we used in our large scale experiments, measured on IBM Blue Horizon. In addition to the 64-processor running times, the table shows the average time per atomic task and the maximum time per atomic task. Minimum times are on the order of several milliseconds; thus, the variance in atomic task sizes is huge. For graph 37e, the average time per task decreases by over 50% when the dynamic depth increment is changed from 1 to 2 — this is indeed the desired effect of splitting up large pieces.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Increment</th>
<th>Completion time ( (T_{4}) )</th>
<th>Average time/task</th>
<th>Maximum time/task</th>
</tr>
</thead>
<tbody>
<tr>
<td>35k</td>
<td>1</td>
<td>13.31 h</td>
<td>222 s</td>
<td>8.27 h</td>
</tr>
<tr>
<td>36c</td>
<td>2</td>
<td>21.87 h</td>
<td>177 s</td>
<td>9.18 h</td>
</tr>
<tr>
<td>37e</td>
<td>1</td>
<td>13.88 h</td>
<td>430 s</td>
<td>6.93 h</td>
</tr>
<tr>
<td>37e</td>
<td>2</td>
<td>12.39 h</td>
<td>200 s</td>
<td>6.57 h</td>
</tr>
</tbody>
</table>

Table 1: Input Graph Parameters for TSP.

4.1.2 The Effect of Dynamic Depth Expansion

To demonstrate the effect of our dynamic depth expansion, described in Section 3.2.3, we varied the depth increment parameter from 0 through 5 for a given test graph. Figure 10 shows a set of speedup curves for a smaller, 32-node graph, run on the Paderborn Linux cluster. Although these are speedup curves in the classical sense, the base case is not measured on a single processor. Instead, we show speedup over \( T_{4} \), the time it took 4 processors to compute the test graph. The reasons for this are twofold:

1. The time to complete a run with 4 processors is approximately 3 hours. The time to run on a single processor is therefore prohibitively long, and since speedup in that region of the curve is close to optimal, does not add anything qualitatively.

2. In taking a 4-processor experiment as our base case, we eliminate some of the potential for observing super-linear speedup by adding a breadth-first component to our local depth-first algorithm.

The results show that speedup was mediocre with 0 increment (\( dinc0 \) in the diagram), i.e., with a static atomic task size. This is due to a single large piece in this graph that imbalances the load and lengthens the critical path. With an increment of 1 per round, the result is improves. Increments of 2 and 3 yield more speedup, although the improvement is not so dramatic. The best results were obtained with an increment of 4 for this graph, at least until the saturation point was reached at 64 processors and the overall running time became too short. An increment of 5 performed much worse, about the same as 1. This is due to the large number of tasks per round generated by such high increments, placing an additional burden on the client process and the communication system. Other experiments with different graphs generally confirmed the following observations:

- For each graph, there is an individual optimum depth increment, usually around 2 or 3.

- The more imbalance a graph shows with 0 increment, the more likely it is that a higher increment yields a better result.

- Dynamic depth expansion is most useful in the center of the curve, when overall running times are still long enough to split up large pieces. For the smaller host configurations, it is often not needed to achieve good results; in the very largest configurations, the short total running time means that some remaining large pieces will still prolong the critical path.

In summary, this feature has greatly improved the system’s performance, and enables good speedups for graphs that were previously deemed unsuitable.

4.1.3 Large Scale Experiments

The next set of results, shown in Figure 11, were run on the IBM Blue Horizon. Here, we chose a base case of 64 processors, for the same reasons as stated above. The results show good speedup up to around 500 processors. Again, we
can show the benefit of dynamic depth expansion: for graph 37e, with an increment of 1, speedup dropped off strongly at 500 processors. With an increment of 2, the result was better in every configuration. Interestingly, our best result, for graph 35k, was achieved with depth increment 1 — the graph showed no further improvement for higher increments, which means that it was evenly balanced to begin with. The largest measurements, on up to 1024 processors, show some dropoff in speedup and efficiency. We attribute this to the overall increase in communication-to-computation ratio: the total time to compute the problem with 1024 hosts was between 40 minutes and 2 hours, depending on the graph. Our experience shows that communication overhead becomes significant when overall running time drops below one hour. Also, any large pieces not yet split up by dynamic depth expansion lengthen the critical path. At this point, we are convinced that the system is not at fault — if we increase the size of the test graph, we should see better speedup. This, however, was not possible with the current allocation of 20,000 units on Blue Horizon. Instead, we would require around 80,000 units for such large tests.

![Graph 35K Speedup](image)

**Figure 11:** Speedup over T64 on IBM Blue Horizon.

Overall, these results are much improved, compared to earlier versions of Javelin: Due to algorithmic improvements, faster processors, and the advent of a faster JVM in the JDK 1.3 release, we are able to test graphs whose search trees are several orders of magnitude larger, even though communication latency remains the same. Scalability is significantly higher, too: we were able to fully utilize up to 1024 processors on the IBM Blue Horizon machine, without reaching an obvious scalability limit of our software.

### 4.2 Fault Tolerance Experiments

We present some experiments designed to validate our eager scheduling analysis. In these tests, we run a raytracing application: a classical example of a master-worker computation; its atomic task running time variance is relatively low; in a branch-and-bound example like the TSP, running times vary significantly. High variance in the running time of atomic tasks violates the first of assumption in Section 3.4: the running time of atomic tasks \( \Delta t \) is not negligible in comparison to the total compute time of the problem \( T \).

In these tests, we rendered a scene of 5223 objects on up to 10 hosts, using the 400 MHz dual processor nodes of our local Linux cluster. Figure 12 compares running times with and without induced failures. The right columns show failure-free runs of 9, 8, 7, and 6 hosts. The left columns show results of tests in which we killed a number of hosts at times symmetric to the computation’s halfway point. In the leftmost experiment, we started 10 hosts, and killed 2 during the computation. According to our analysis, \( T_{10}(2) \) should equal \( T_5 \); the measured result was within 3% of this prediction. The next result, \( T_{10}(4) \), was within 4% of the predicted value, \( T_8 \). \( T_{10}(6) \) was approximately 6% slower than its predicted value, \( T_{10} \). Finally, \( T_{10}(8) \), a test in which we killed 8 out of 10 hosts during the computation, was within 7.5% of the target, \( T_6 \).

The increasing gap between the times with and without failures can be explained by the non-negligible loss of work in this scenario: The average running times of the atomic tasks were about 27 secs, and maximum times were as large as 57 secs. The more hosts are killed, the more work needs to be rescheduled and redistributed; a small overhead is to be expected. Overall, we consider these results a confirmation of our analysis.

![Fault Tolerance](image)

**Figure 12:** Fault Tolerance on PII 400 Processors.

### 5. CONCLUSION

Javelin 3 harvests unused machine cycles of networked computers for ultra-large, coarse-grained *adaptively parallel* applications. It runs well on large cluster machines and networked workstations, as long as the ubiquitous Java platform is installed. We use work stealing, integrated with an advanced form of eager scheduling, to balance the computational load and achieve fault tolerance and scalability. The TSP branch-and-bound application is a stress test of our system, because the computational load of tasks at the same depth in the search tree can vary from as small as less than 1 millisecond to as large as more than half the time to complete the entire computation. To cope with this computational load variance, the eager scheduler can dynamically decrease the size parameter of an atomic task. This new form of eager scheduling substantially improves the system’s performance: We can now solve much larger problems than before, and also obtain better speedups for problems that previously did not scale well.

Performance for our test graphs show near ideal speedups through 256 hosts. When 512 are used, speedup tapers off,
but only when 1,024 hosts are used do speedups reduce substantially. With larger problem instances (e.g., those that take 256 processors 20 or more hours to complete) we expect higher speedups even using 1,024 or more processors. We believe that the principle impediment to better speedups is the short running time using 1,024 hosts: Large tasks not broken up by the eager scheduler prolong the critical path.

The quantification of performance degradation due to host failures that our model predicts is born out by the measurements taken of the raytracing application. The model thus enables users to predict their application’s performance as a function of host failure rates, as long as the model’s assumptions remain valid.

6. REFERENCES


