A Development and Deployment Framework for Distributed Branch & Bound

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

May 1, 2007

1 Introduction

Branch-and-bound intelligently searches the set of feasible solutions to a combinatorial optimization problem: It, in effect, proves that the optimal solution is found without necessarily examining all feasible solutions. The feasible solutions are not given. They can be generated from the problem description. However, doing so usually is computationally infeasible: The number of feasible solutions typically grows exponentially as a function of the size of the problem input. For example, the set of feasible tours in a symmetric Traveling Salesman Problem (TSP) of a complete graph with 23 nodes is 22!/2 or around 8×10^{14} tours. The space of feasible solutions is progressively partitioned (branching), forming a search *tree*. Each tree node has a partial feasible solution. The node *represents* the set of feasible solutions that are extensions of its partial solution. For example, in a TSP branch and bound, a search tree node has a partial tour, representing the set of all tours that contain that partial tour. As branching continues (progresses down the problem tree), each search node has a more complete partial solution, and thus represents a smaller set of feasible solutions. For example, in a TSP branch and bound, a tree node's children each represent an extension of the partial tour to a more complete tour (e.g., one additional city or one additional edge). As one progresses down the search tree, each node represents a larger partial tour. As the size of a partial tour increases, the number of full tours containing the partial tour clearly decreases.

In traversing the search tree, we may come to a node that represents a set of feasible solutions, all of which are provably more costly than a feasible solution already found. When this occurs, we *prune* this node of the search tree: We discontinue further exploration of this set of feasible solutions. In the example of the TSP problem, the cost of any feasible tour that has a given partial tour surely can be bounded from below by the cost of the partial tour: the sum of the edge weights for the edges in the partial tour. (In our experiments, we use a Held-Karp lower bound, which is stronger but more computationally complex.) If the lower bound for a node is *higher* than the current upper bound (i.e., best known complete solution), then the cost of all complete solutions (e.g., tours) represented by the node is higher than a complete solution that is already known: The node is pruned. Please see Papadimitriou and Steiglitz [23] for a more complete discussion of branch-and-bound. Fig. 8 gives a basic, sequential branch-and-bound algorithm.

The framework that we present here is designed for deployment in a distributed setting. Moreover, the framework supports *adaptive parallelism*: During the execution, the set of compute servers can grow (if new compute servers become available) or shrink (if compute servers become unavailable or fail): The branch and bound computation thus cannot assume a fixed number of compute servers.

The branch and bound computation is decomposed into tasks, each of which is executed on a compute server: Each element of the active set (please see Fig. 8) is a task that, in principle, can be scheduled for execution on any compute server. Indeed, parallel efficiency requires load balancing of tasks among compute servers. This distributed setting implies the following compute server requirements:

- Tasks (activeset elements) are generated during the computation — they cannot be scheduled *a priori*;

- When a compute server discovers a new best cost, it must be propagated to the other compute servers;
- Detecting termination requires "knowing" when all branches (children) have been either fully examined or pruned. In a distributed setting, the implied communication must not be a bottleneck.

Our goal is to facilitate the development of branch and bound computations for deployment as a distributed computation. We provide a development-deployment infrastructure that requires the developer to write code for *only* the particular aspects of the branch and bound computation under development, primarily the branching rule, the lower bound computation, and the upper bound computation. We present this framework and some experimental results of its application to a medium complexity Traveling Salesman Problem (TSP) code running on a beowulf cluster.

2 Related Work

Held, Hoffman, Johnson, and Wolfe give a short history of the Traveling Salesman Problem [10]. In it, they note that, in 1963, Little, Murty, Sweeney, and Karel [17] were the first to use the term "branch and bound" to describe their enumerative procedure for solving TSP instances. As we understand it, Little et al. and Land and Doig [16] independently discovered the technique of branch and bound. This discovery led to "a decade of enumeration".

Parallel branch and bound also has been widely studied. See, for example, [28, 29]. Rather early on it was discovered that there are speedup anomalies in parallel branch and bound [15]: Completion times are not monotonically non-increasing as a function of the number of processors. In the discussion that follows, let T denote the search tree, c^* denote the cost of a minimum cost leaf in T, $T^* \subseteq T$ denote the subtree of T whose nodes cost less than or equal to c^*, n denote the number of nodes in T^*, and h denote the height of T^*. In [13], Karp and Zhang present a universal randomized method called Local Best-First Search for parallelizing sequential branch-and-bound algorithms. When executing on a completely connected, message-passing multiprocessor, the method's computational complexity is asymptotically optimal with high probability: $O(n/p+h)$, where p is the number of processors. The computational complexity of maintaining the local data structure and the communication overhead are ignored in their analysis. When $n > p^2 \log p$, Liu, Aiello, and Bhatt [18] give a method for branch and bound that is asymptotically optimal with high probability, assuming that interprocessor communication is controlled by a central FIFO arbiter. Herley, Pietracaprina, and Pucci [11], give a deterministic parallel algorithm for branch and bound based on the parallel heap selection algorithm of Frederickson [9], combined with a parallel priority queue. The complexity of their method is $O(n/p + h \log^2(np))$ on an EREW-PRAM, which is optimal for $h = O(n/(p \log^2(np)))$. This bound includes communication costs on a EREW-PRAM.

Distributed branch and bound also has been widely studied. Tschoeke, Lueling, and Monien contributed experimental work on distributed branch and bound for TSP [27] using over 1,000 processors. When the number of processors gets large, fault tolerance becomes an issue. Yahfoufi and Dowaji [31] present perhaps the first distributed fault-tolerant branch and bound algorithm.

There also has been a lot of work on what might be called programming frameworks for distributed branch and bound computation. This occurs for two reasons: 1) branch and bound is best seen as a meta-algorithm for solving large combinatorial optimization problems: It is a framework that must be completed with problem-specific code; 2) programming a fault tolerant distributed system is sufficiently complex to motivate a specialization of labor: distributed systems research vs. operations research. In 1995, Shinano et al. [26] presented PUBB, a Parallel Utility for Branch-and-Bound, based on the C programming language. They illustrate the use of their utility on TSP and 0/1 ILP. They introduce the notion of a Logical Computing Unit (LCU). Although in parts of their paper, an LCU sounds like a computational task, we are persuaded that it most closely resembles a processor, based on their explanation of its use: "The master process maintains in a queue, all the subproblems that are likely to lead to an optimal solution. As long as this queue is not empty and an idle LCU exists, the master process selects subproblems and assigns them to an idle LCU for evaluation one after the other." When discussing their experimental results, they note "The results indicate that, up to using about 10 LCUs, the execution time rapidly decreases as more LCUs are added. When the number of LCUs exceeds about 20, the computing time for one run, remains almost constant." Indeed, from their Fig. 9, we can see that PUBB's parallel efficiency steadily goes down when the number of LCUs is above 10, and is well below 0.5, when the number of LCUs is 55. Aversa et al. [3] report on a the Magda project for mobile agent programming with parallel skeletons. Their divide-and-conquer skeleton is used to implement branch and bound, which they provide experimental data for on up to 8 processors. Moe [20] reports on GRIBB, and infrastructure for branch and bound on the Internet. Experimental results on an SGI Origin 2000 with 32 processors machines shows good speedups when the initial bound is tight, and about 67% of ideal speedup, when a simple initial bound is used. Dorta et al. [8] present C++ skeletons for divide-and-conquer and branch-and-bound, where deployment is intended for clusters. Their experiments, using a 0/1 Knapsack problem of size 1000. On a Linux cluster with 7 processors, the average speedup was 2.25. On an Origin 3000 with 16 processors, the average speedup was 4.6. On a Cray T3E with 128 processors, the average speedup was 5.02. They explain "Due to the fine grain of the 0/1 knapsack problem, there is no lineal increase in the speed up when the number of processor increase. For large numbers of processors the speed up is poor."

Neary, Phipps, Richman, and Cappello [22, 21] present an infrastructure/framework for distributed computing, including branch and bound, that tolerates faulty compute servers, and is in pure Java, allowing application codes to run on a heterogeneous set of machine types and operating systems. They experimentally achieved about 50% of ideal speedup for their TSP code, when running on 1,000 processors. Their schemes for termination detection and fault tolerance of a branch and bound computation both exploit its *tree-structured* search space. The management of these schemes is centralized. Iamnitchi and Foster [12] build on this idea of exploiting branch and bound's tree-structured search space, producing a branch and bound-specific fault tolerance scheme that is distributed, although they provide only simulation results.

3 The Deployment 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 [24, 25] 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 associated with 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: A host (aka compute server) joins a particular task server. Once joined, 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 80.2.

Task objects *encapsulate* computation: Their inputs and outputs are managed by Jicos. 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.

3.1 Tolerating Faulty Hosts

To support self-healing, all proxy objects are leased [19, 2]. 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 [4], such as Satin [30], need to recompute some task decomposition computations, even if they completed successfully. In some applications, such as sophisticated TSP codes, decomposition can be computationally complex. On Jicos, only the task that was currently being executed needs to be recomputed. This is a substantial improvement. In the TSP problem 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 Considerations

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 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, 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 more 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.

5 The Computational Model

Computation is modeled 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. A computation's tasks all have access to an *environment* consisting of an immutable *input* object and a mutable *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.

6 The Branch and Bound API

Tasks correspond to nodes in the search tree: Each task gives rise to a set of smaller subtasks, until it represents a node in the search tree that is small enough to be explored by a single compute server. We refer to such a task as *atomic*; it does not decompose into subtasks.

6.1 The Environment

For branch and bound computations, the environment *input* is set to the problem instance. For example, in a TSP, the input can be set to the distance matrix. Doing so materially reduces the amount of information needed to describe a task, which reduces the time spent to marshal and unmarshal such objects.

The cost of the least cost known solution at any point in time is shared among the tasks: It is encapsulated as the branch and bound computation's *shared* object. (Please see IntUpperBound below.) In branch and bound, this value is used to decide if a particular subtree of the search tree can be pruned. Thus, sharing the cost of the least cost known solution enhances the pruning ability of concurrently executing tasks that are exploring disjoint parts of the search tree. Indeed, this improvement in pruning is essential to the efficiency of parallel branch and bound. When a branch and bound task finds a complete solution whose cost is less than the current least cost solution, it sets the shared object to this new value, which implicitly causes JICOS to propagate the new least cost throughout the distributed system.

6.2 The JICOS Branch and Bound Framework

The classes comprising the JICOS branch and bound framework are based on 2 assumptions:

- The branch and bound problem is formulated as a minimization problem. Maximization problems typically can be reformulated as minimization problems.
- The cost can be represented as in int.

Should these 2 assumptions prove troublesome, we will generalize this framework.

Before giving the framework, we describe the problem-specific class that the *application developer must provide*: A class that implements the *Solution* interface. This class represents nodes in the search tree: A Solution object is a partial feasible solution. For example, in a TSP, it could represent a partial tour. Since it represents a node in the search tree, its children represent more complete partial feasible solutions. For example, in a TSP, a child of a Solution object would represent its parent's partial tour, but including[excluding] one more edge (or including one more city, depending on the branching rule).

The Solution interface has the following methods:

getChildren returns a queue of the Solution objects that are the children of this Solution. The queue's retrieval order represents the application's selection rule, from most promising to least promising. In particular, the first child is cached (please see § 4 for an explanation of task caching).

getLowerBound returns the lower bound on the cost of any complete Solution that is an extension of this partial Solution.

getUpperBound returns an upper bound on the cost of any complete Solution, and enables an upper bound heuristic for incomplete solutions.

isComplete returns true if and only if the partial Solution is, in fact, complete.

reduce Omit loose constraints. For example, in a TSP solution, this method may omit edges whose cost is greater than the current best solution, and therefore cannot be part of any better solution. This method returns void, and can have an empty implementation.

The classes that comprise the branch and bound framework—*provided by* JICOS to the application programmer—are described below:

BranchAndBound This is a Task class, which resides in the jicos.applications.branchandbound package, whose objects represent a search node. A BranchAndBound Task either:

- constructs smaller BranchAndBound tasks that correspond to its children search nodes, or
- fully searches a subtree, returning:
  - null, if it does not find a solution that is better than the currently known best solution
  - the best solution it finds, if it is better than the currently known best solution.

IntUpperBound A object that represents the minimum cost among all known complete solutions. This class is in the jicos.services.shared package. It implements the Shared interface (for details about this interface, please see the JICOS API), which defines the shared object. In this case, the shared object is an Integer that holds the current upper bound on the cost of a minimal solution. Consequently, IntUpperBound u "is newer than" IntUpperBound v when $u < v$.

MinSolution This task is included in the jicos.services.tasks package. It is a composition class whose execute method:

- receives an array of Solution objects, some of which may be null;
- returns the one whose cost is minimum, provided it is less than or equal to the current best solution. Equality is included to ensure that the minimum cost *solution* is reported: It is not enough just to know the *cost* of the minimum cost solution.
- From the standpoint of the JICOS system (not a consideration for application programming), the compose tasks form a tree that performs a gather operation, which, in this case, is a min operation on the cost of the Solution objects it receives. Each task in this gather tree is assigned to some task server, distributing the gather operation throughout the network of task servers. (This task is indeed executed on a task server, not a compute server - please see § 4.)

Q A queue of Solution objects.

Using this framework, it is easy to construct a branch and bound computation. The JICOS web site tutorial [1] illustrates this, giving a complete code for a simple TSP branch and bound computation.

7 Experimental Results

7.1 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.

7.2 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 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 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.

7.3 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 compute server was started on its own machine. Except for 28 compute servers in the 120 processor case (which were calibrated with a separate base case), each compute server 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.

One 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 speedup formula a heterogeneous processor set [7]. We thus 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 server. We recorded the completion times and report the averages of 8 runs.

7.4 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. It is well known [4] that $\max\{T_\infty, T_1/P\} \leq T_P$. Thus, $0 \leq \max\{T_\infty, T_1/P\}/T_P \leq 1$ is a *lower bound* on the fraction of perfect speedup that is actually attained. Figure 80.3 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 0.94; 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 just 11 minutes!

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 [14]. 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 [6, 7] 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.

Our fault tolerance data is summarized in Table 80.1. Overhead is caused by the rescheduling of tasks lost when a compute server failed as well as some time taken by the TaskServer to recognize a faulty compute server. Negative overhead is a consequence of network traffic and thread scheduling preventing a timely transfer of the kill command to the appropriate compute server.

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.

8 Conclusion

We have presented a framework, based on the JICOS API, for developing distributed branch and bound computations. The framework allows the application developer to focus on the problem-specific aspects of branch and bound: the lower bound computation, the upper bound computation, and the branching rule. Reducing the code required to these problem-specific components reduces the likelihood of programming errors, especially those associated with distributed computing, such as threading errors, communication protocols, and detecting, and recovering from, faulty compute servers.

The resulting application can be deployed as a distributed computation via JICOS running, for example, on a beowulf cluster. JICOS [5] scales efficiently as indicated by our speedup experiments. This, we believe, is because we have carefully provided 1) for divide-and-conquer computation; 2) an environment that is common to all compute servers for computation input (e.g., a TSP distance data structure, thereby reducing task descriptors) and a mutable shared object that can be used to communicate upper bounds as they are discovered; 3) latency hiding techniques of task caching and pre-fetching; and 4) latency reduction by distributing termination detection on the network of task servers.

Faulty compute servers are tolerated with high efficiency, 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). Finally, the overhead of task servers is shown to be quite small, further confirming the efficiency of JICOS as a distributed system.

The vast majority of the code concerns JICOS, the distributed system of fault tolerant compute servers. The Java classes comprising the branch-and-bound framework are few, and easily enhanced, or added to, by operations researchers; the source code is cleanly designed and freely available for download from the JICOS web site [1].

References

[1] JICOS: A Java-Centric Network Computing Service. Web site. http://cs.ucsb.edu/projects/jicos.

[2] K. Arnold. *The Jini specifications.* Addison-Wesley, 2 edition, 1999.

[3] R. Aversa, B. D. Martino, N. Mazzocca, and S. Venticinque. Mobile Agent Programming for Clusterswith Parallel Skeletons. In J. Palma, J. Dongarra, V. Hernndez, and A. A. Sousa, editors, *Proc. High Performance Computing for Computational Science - VECPAR 2002: 5th International Conference*, pages 622 – 634, June 2002. Lecture Notes in Computer Science, Springer-Verlag.

[4] 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.

[5] P. Cappello and C. J. Coakley. JICOS: A Java-Centric Network Computing Service. In *The Proceedings of the 17th IASTED Int. Conference on Parallel and Distributed Computing and Systems*, Nov. 2005. accepted.

[6] 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.

[7] 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.

[8] I. Dorta, C. Leon, C. Rodriguez, and A. Rojas. Parallel Skeletons for Divide-and-Conquer and Branch-and-Bound Techniques. In *Proc. of the 11th Euromicro Conf. on Parallel, Distributed and Network-Based Processing (Euro-PDP'03)*, 2003. IEEE Computer Society.

[9] G. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104:197–214, 1993.

[10] M. Held, A. J. Hoffman, E. L. Johnson, and P. Wolfe. Aspects of the traveling salesman problem. *IBM J. Res. Development*, 28(4):476–486, July 1984.

[11] K. Herley, A. Pietracaprina, and G. Pucci. Fast Deterministic Parallel Branch-and-Bound. *Parallel Proc. Lett.*, 1999.

[12] A. Iamnitchi and I. Foster. A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems. In *Proceedings of the 2000 International Conference on Parallel Processing*, August 2000. Toronto, Canada.

[13] R. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 290–300, New York, NY, USA, 1988. ACM Press.

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

[15] T.-H. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Commun. ACM*, 27(6):594–602, 1984.

[16] A. H. Land and A. G. Doig. An Automatic Method for Solving Discrete Programming Problems. *Econometrica*, 28:497–520, 1960.

[17] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An Algorithm for the Traveling Salesman Problem. *Oper. Res.*, 11:972–989, 1963.

[18] P. Liu, W. Aiello, and S. Bhatt. An atomic model for message-passing. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 154–163, New York, NY, USA, 1993. ACM Press.

[19] M. S. Miller and K. E. Drexler. Markets and Computation: Agoric Open Systems. In B. Huberman, editor, *The Ecology of Computation.* Elsevier Science Publishers B. V., North-Holland, 1988.

[20] R. Moe. GRIBB Branch-and-Bound Methods on the Internet. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and et al., editors, *Proc. Parallel Processing and Applied Mathematics: 5th International Conference, PPAM, Workshop on High Perfomance Numerical Algorithms*, pages 1020 – 1027, September 2003. Lecture Notes in Computer Science, Springer-Verlag.

[21] 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.

[22] 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. Postprint available free at: http://repositories.cdlib.org/postprints/40.

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

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

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

[26] Y. Shinano, M. Higaki, and R. Hirabayashi. A generalized utility for parallel branch and bound algorithms. In *Proc. 7th Int. Symp. on Parallel and Distributed Processing*, page 392, 1995. IEEE.

[27] S. Tschke, R. Lling, and B. Monien. Solving the Traveling Salesman Problem with a Distributed Branch-and-Bound Algorithm on a 1024 Processor Network. In *Proc. of the 9th International Parallel Processing Symposium*, pages 182–189, 1995. IEEE Computer Society Press.

[28] B. W. Wah, G. J. Li, and C. F. Yu. Multiprocessing of combinatorial search problems. *IEEE Computers*, June 1985.

[29] B. W. Wah and C. F. Yu. Stochastic modeling of branch-and-bound algorithms with best-first search. *IEEE Transactions on Software Engineering*, SE-11:922–934, Sept. 1985.

[30] 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.

[31] N. Yahfoufi and S. Dowaji. Self-stabilizing distributed branch-and-bound algorithm. In *Proceedings of the 1996 IEEE 15th Annual International Phoenix Conference on Computers and Communications*, pages 246–252, 1996.

```
activeSet = { originalTask };
u = infinity; // u = the cost of the best solution known
currentBest = null;
while (  ! activeSet.isEmpty() ) {
    k = remove some element of the activeSet;
    children = generate k's children;
    for each element of children {
        if ( element's lower bound <= u )
            if ( element is a complete solution ) {
                u = element's cost;
                currentBest = element;
            }
            else
                add element to activeSet;
    }
}
```

Figure 80.1: A sequential algorithm for branch and bound.

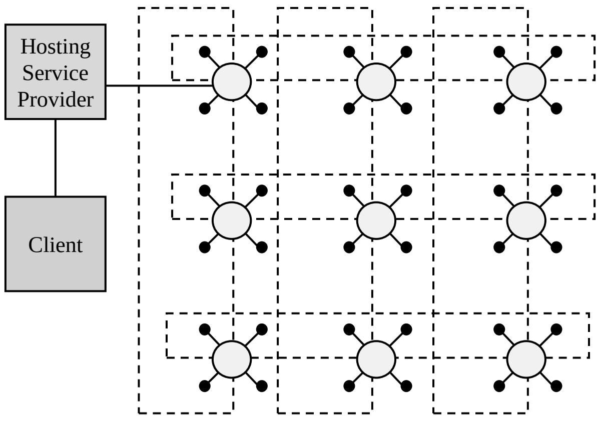

Figure 80.2: A Jicos system that has 9 task servers. The task server topology, a 2D torous, is indicated by the dashed lines. In the figure, each task server has 4 associated hosts (the little black discs). An actual task server might serve about 40 hosts (although our experiments indicate that 128 hosts/task server is not too much). The client interacts *only* with the HSP.

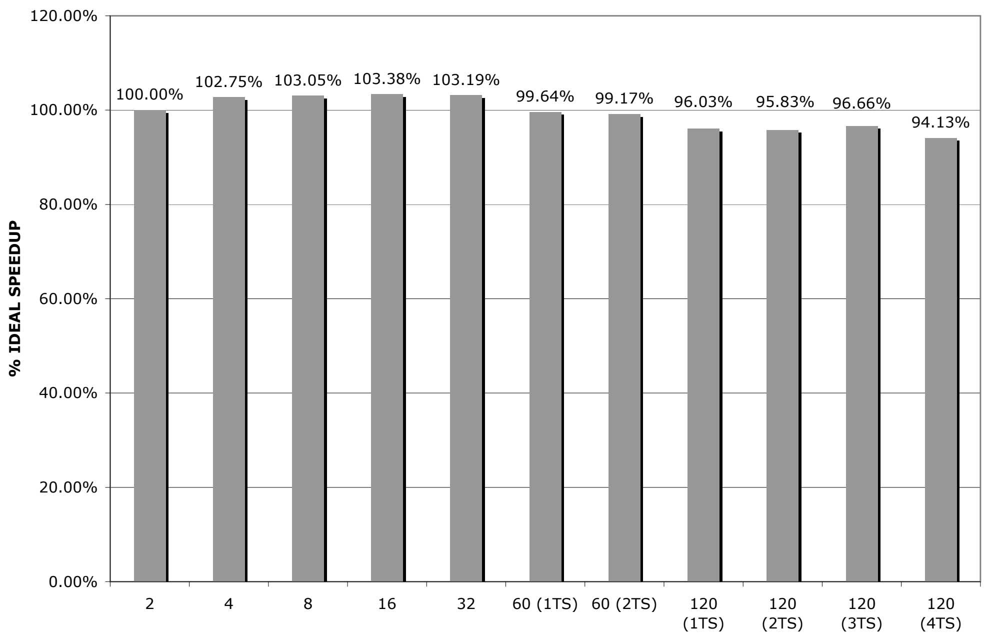

Figure 80.3: Number of processors vs. % of ideal speedup.

Table 80.1: Efficiency of compute server fault tolerance. Each experiment started with 32 processors. The experiment in which 30 processors finished had 2 fail; the experiment in which 4 finished had 28 fail.

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

Index

adaptive parallelism, 2

branch and bound
    distributed, 4
    framework, 4, 9, 10
    history, 3
    parallel, 3
    sequential, 19
    termination, 2

Environment, 8, 9

fault tolerance, 7
    recomputation, 7

Host, 5, 6
Hosting Service Provider, 6

Input, 8, 9

lower bound
    Held-Karp , 2

parallelism
    maximum, 13

search tree, 1, 9
self-healing, 7
Shared, 8, 9, 11
Solution interface, 10, 11
speedup
    heterogeneous, 13
    homogeneous, 13

Task, 2, 6, 10, 11
    atomic, 9
    caching, 7
    Executed by task server, 8
    prefetching, 7
Task server, 6
Traveling salesman problem, 1
    TSPLIB, 12