

41

A Development and Deployment Framework for Distributed Branch and Bound

Peter Cappello

University of California

Christopher James Coakley

University of California

41.1	Introduction	41-1
41.2	Related Work	41-2
41.3	The Deployment Architecture	41-4
	Tolerating Faulty Hosts	
41.4	Performance Considerations	41-5
41.5	The Computational Model	41-6
41.6	The Branch-and-Bound API	41-6
	The Environment • The JICOS Branch-and-Bound Framework	
41.7	Experimental Results	41-8
	The Test Environment • The Test Problem •	
	The Measurement Process • The Measurements	
41.8	Conclusion	41-10

41.1 Introduction

Branch-and-bound ~~algorithm~~ 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 represents 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

```

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 41.1 A sequential algorithm for branch and bound.

of the TSP, 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 (see Papadimitriou and Steiglitz [1] for a more complete discussion of branch-and-bound). Figure 41.1 gives a basic, sequential branch-and-bound algorithm.

Branch-and-bound algorithms may be easily modified to generate suboptimal solutions. The total search time decreases as the desired accuracy decreases.

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 (see Figure 41.1) 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 TSP code running on a beowulf cluster.

41.2 Related Work

Held et al. give a short history of the TSP [2]. In it, they note that, in 1963, Little et al. [3] 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. [3] and Land and Doig [4] independently discovered the technique of branch and bound. This discovery led to “a decade of enumeration.”

Parallel branch and bound has also been widely studied. See, for example, Refs. [5,6]. Rather early on it was discovered that there are speedup anomalies in parallel branch and bound [7]: Completion times are not monotonically nonincreasing as a function of the number of processors. In the discussion that follows, let T denote the search tree, c^* the cost of a minimum-cost leaf in T , $T^* \subseteq T$ the subtree of T whose nodes cost less than or equal to c^* , n the number of nodes in T^* , and h the height of T^* . Karp and Zhang [8] 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 is ignored in their analysis. When $n > p^2 \log p$, Liu et al. [9] 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 et al. [10] give a deterministic parallel algorithm for branch and bound based on the parallel heap selection algorithm of Frederickson [11], 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 an EREW-PRAM.

AQ1

Distributed branch and bound has also been widely studied. Tschöke et al. [12] contributed experimental work on distributed branch and bound for TSP using over 1000 processors. When the number of processors gets large, fault tolerance becomes an issue. Yahfoufi and Dowaji [13] 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 metaalgorithm for solving large combinatorial optimization problems: It is a framework that must be completed with problem-specific code and (2) programming a fault-tolerant distributed system is sufficiently complex to motivate a specialization of labor: distributed systems research versus operations research. In 1995, Shinano et al. [14] presented a Parallel Utility for Branch and Bound (PUBB) 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 Figure 9 (in Ref. [14]), 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. [15] report on 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 up to eight processors. Moe [16] reports on GRIBB, and infrastructure for branch and bound on the Internet. Experimental results on an SGI Origin 2000 with 32 processors machines show good speedups when the initial bound is tight, and $\sim 67\%$ of ideal speedup, when a simple initial bound is used. Dorta et al. [17] 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 seven 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 et al. [18] and Neary and Cappello [19] 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 50% of ideal speedup for their TSP code, when running on 1000 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 [20] 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.

41.3 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 [21,22] front end 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 three service component classes.

Hosting Service Provider. JICOS clients (i.e., processes seeking computation done on their behalf) interact solely with the hosting service provider (HSP) component. A client logs in, submits computational tasks, requests results, and logs out. When interacting with a client, the HSP 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 HSP. 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, which 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 log-in to all task servers, who in turn propagate it to all their hosts. When a client logs out, the HSP propagates that log-out 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 log-in and log-out, are transmitted via a task server *tree*: a subgraph of the torous (see Figure 41.2).

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 prefetching, and task execution on task servers for selected task classes.

41.3.1 Tolerating Faulty Hosts

To support self-healing, all proxy objects are leased [23,24]. 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

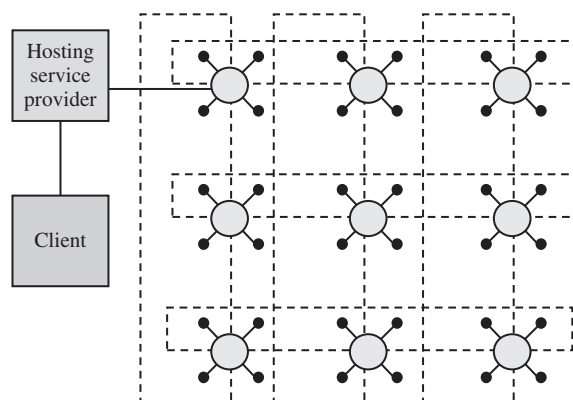


FIGURE 41.2 A JICOS system that has nine task servers. The task server topology, a 2D torus, is indicated by the broken lines. In the figure, each task server has four associated hosts (the little black discs). An actual task server might serve about 40 hosts (although our experiments indicate that 128 hosts/task server is not too much). The client interacts *only* with the HSP.

minimized: Systems that support divide-and-conquer but do not use explicit continuation passing [25], such as Satin [26], 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 instance that we use for our performance experiments, the average task time is 2 s. Thus, the recomputation time for a failed host is, in this instance, a mere 1 s, on average.

41.4 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 prefetching. The *application* can help hide communication latency via task prefetching:

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 prefetches 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 it knows that it is not going to construct any subtasks, it can invoke its environment's *prefetch* method. This causes its host to request a task from the task server in a separate thread.

Task prefetching overlaps the host's execution of the current task with its request for the next task. Application-directed prefetching, 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 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-TaskServer communication.

41.5 The Computational Model

Computation is modeled by a *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.

41.6 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.

41.6.1 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 (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.

41.6.2 The JICOS Branch-and-Bound Framework

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

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

Should these two 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 (see Section 41.4 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. An 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, 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—see Section 41.4.)

Q A queue of *Solution* objects Using this framework, it is easy to construct a branch-and-bound computation. The JICOS web site tutorial [27] illustrates this, giving a complete code for a simple TSP branch-and-bound computation.

41.7 Experimental Results

41.7.1 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 2.6 GHz (or 3.0 GHz) Xeon processor with 3 GB (2 GB) of PC2100 memory, two 36 GB (32 GB) SCSI-320 disks with on-board controller, and an on-board 1 GB ethernet adapter. The machines are connected via the gigabit link to one of the two 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.

41.7.2 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 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 s, and exactly 30,647 MinSolution tasks whose average execution time was <1 ms.

41.7.3 The Measurement Process

For each experiment, an HSP 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 two hyperthreaded processors that are presented to the JVM as four 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 eight times (or more) and averages were reported.

One processor in the OS does not correspond to one physical processor. It is, therefore, difficult to get meaningful results for one processor. We, consequently, use one machine, which is two local CPUs, as our base case. For the 120 processor measurements, we used the speedup formula of a heterogeneous processor set [28]. We thus had three 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 1500 s, $\sim 3/4$ through the computation. The completion time for the total computation was recorded, and was compared with 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 reported the averages of eight runs.

41.7.4 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 s. It is well known [25] 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.

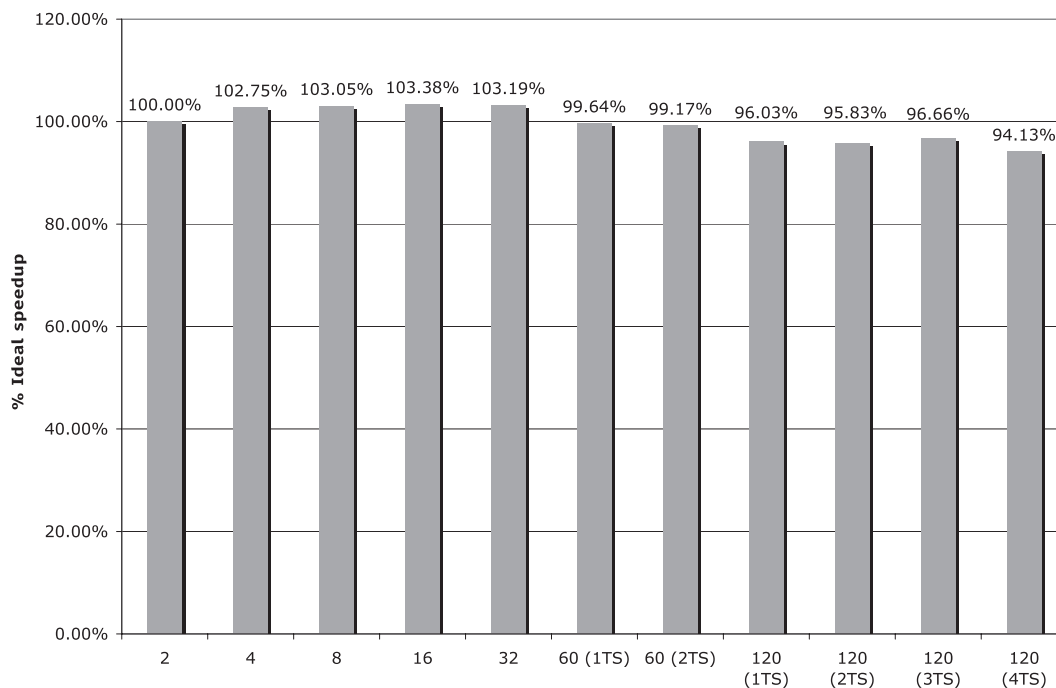


FIGURE 41.3 Number of processors versus % of ideal speedup.

Figure 41.3 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 two-processor base case ran in 9 h and 33 min; whereas the 120-processor experiment (two processors per host) ran in just 11 min!

We get superlinear speedups for 4, 8, 16, and 32 processors. The standard deviation was $<1.6\%$ 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 [29]. 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-s tasks. Even CX [28,30] 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, $P_\infty = 1857$ processors. Thus, 1857 processors is a lower bound on the number of processors necessary to bring the completion time down to T_∞ , namely, 37 s.

Our fault tolerance data is summarized in Table 41.1. 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 s for a dedicated task server and 3114.8 s 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.

TABLE 41.1 Efficiency of Compute Server Fault Tolerance

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

Note: 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.

41.8 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 [31] 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 prefetching; 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 [27]. Our branch-and-bound framework may be used for ~~almost~~ any divide-and-conquer computation. JICOS may be ~~easily~~ adapted to solve in a distributed environment any algorithm that can be defined as a computation over directed acyclic graph, where the nodes refer to computations and the edges specify a precedence relation between computations.

References

- [1] Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
- [2] Held, M., Hoffman, A.J., Johnson, E.L., and Wolfe, P., Aspects of the traveling salesman problem, *IBM J. Res. Dev.*, 28(4), 476, 1984.
- [3] Little, J.D.C., Murty, K.G., Sweeney, D.W., and Karel, C., An algorithm for the traveling salesman problem, *Oper. Res.*, 11, 972, 1963.
- [4] Land, A.H. and Doig, A.G., An automatic method for solving discrete programming problems, *Econometrica*, 28, 497, 1960.
- [5] Wah, B.W., Li, G.J., and Yu, C.F., Multiprocessing of combinatorial search problems, *IEEE Comput.*, 1985.
- [6] Wah, B.W. and Yu, C.F., Stochastic modeling of branch-and-bound algorithms with best-first search, *IEEE Trans. Software Eng.*, SE-11, 922, 1985.

- [7] Lai, T.H. and Sahni, S., Anomalies in parallel branch-and-bound algorithms, *CACM*, 27(6), 594, 1984.
- [8] Karp, R. and Zhang, Y., A randomized parallel branch-and-bound procedure, *Proc. of STOC*, 1988, p. 290.
- [9] Liu, P., Aiello, W., and Bhatt, S., An atomic model for message-passing, *Proc. Symp. on Parallel Algorithms and Architectures*, 1993, p. 154.
- [10] Herley, K., Pietracaprina, A., and Pucci, G., Fast deterministic parallel branch-and-bound, *Parallel Process. Lett.*, 1999.
- [11] Frederickson, G., An optimal algorithm for selection in a min-heap, *Inf. Comput.*, 104, 197, 1993.
- [12] Tschöke, S., Lüling, R., and Monien, B., Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network, *Proc. Int. Parallel Processing Symp.*, 1995, p. 182.
- [13] Yahfoufi, N. and Dowaji, S., Self-stabilizing distributed branch-and-bound algorithm, *Proc. Int. Phoenix Conf. on Computers and Communications*, 1996, p. 246.
- [14] Shinano, Y., Higaki, M., and Hirabayashi, R., A generalized utility for parallel branch and bound algorithms, *Proc. Int. Symp. on Parallel and Distributed Processing*, 1995, p. 392.
- [15] Aversa, R., Martino, S.D., Mazzocca, N., and Venticinque, S., Mobile agent programming for clusters with parallel skeletons, *Proc. Int. Conf. on High Performance Computing for Computational Science*, Lecture Notes in Computer Science, Springer, Berlin, 2002, p. 622.
- [16] Moe, R., GRIBB: Branch-and-bound methods on the Internet, *Proc. Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science, Springer, Berlin, 2003, p. 1020.
- [17] Dorta, I., Leon, C., Rodriguez, C., and Rojas, A., Parallel skeletons for divide-and-conquer and branch-and-bound techniques, *Proc. Euromicro Conf. on Parallel, Distributed and Network-Based Processing*, 2003.
- [18] Neary, M.O., Phipps, A., Richman, S., and Cappello, P., Javelin 2.0: Java-based parallel computing on the Internet, *Euro-Par*, 2000, p. 1231.
- [19] Neary, M.O. and Cappello, P., Advanced eager scheduling for Java-based adaptively parallel computing, *Concurrency Comput.: Pract. Exp.*, 17, 797, 2005.
- [20] Iamnitchi, A. and Foster, I., A problem-specific fault-tolerance mechanism for asynchronous, distributed systems, *Proc. Int. Conf. on Parallel Processing*, 2000.
- [21] Seed, R. and Sandholm, T., A Note on Globus Toolkit 3 and J2EE, Technical report, Globus, 2003.
- [22] Seed, R., Sandholm, T., and Gawor, J., Globus Toolkit 3 Core—a Grid Service Container Framework, Technical report, Globus, 2003.
- [23] Miller, M.S. and Drexler, K.E., Markets and computation: agoric open systems, in *The Ecology of Computation*, Huberman, B., Ed., Elsevier, Amsterdam, 1988.
- [24] Arnold, K., *The Jini Specifications*, 2nd ed. Addison-Wesley, Reading, MA, 1999.
- [25] Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., and Zhou, Y., Cilk: an efficient multithreaded runtime system, *Proc. Symp. on Principles and Practice of Parallel Programming*, 1995, p. 207.
- [26] Wrzesinska, G., van Nieuwpoort, R.V., Maasen, J., Kielmann, T., and Bal, H.E., Fault-tolerant scheduling of fine-grained tasks in Grid environments, *Int. J. High Performance Comput. Appl.*, to appear.
- [27] JICOS: A Java-Centric Network Computing Service, Web site, <http://cs.ucsb.edu/projects/jicos>.
- [28] Cappello, P. and Mourloukos, D., CX: a scalable, robust network for parallel computing, *Sci. Program.*, 10(2), 159, 2001.
- [29] Krintz, C. and Sherwood, T., private communication, 2005.
- [30] Cappello, P. and Mourloukos, D., A scalable, robust network for parallel computing, *Proc. Joint ACM Java Grande/ISCOPE Conf.*, 2001, p. 78.
- [31] Cappello, P. and Coakley, C.J., JICOS: a Java-centric network computing service, *Proc. of the Int. Conf. on Parallel and Distributed Computing and Systems*, 2005.

Query Form

CRC Press

The C-5505 Handbook, Second Edition

JOURNAL TITLE:	84002
ARTICLE NO:	CH041

Queries and / or remarks

Query No.	Details required	Author's response
AQ1	Please expand at the first occurrence.	