A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems

Adriana Iamnitchi
[1]Department of Computer Science
The University of Chicago
Chicago, IL 60637
anda@cs.uchicago.edu

Ian Foster [1,2]
[2]Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
foster@mcs.anl.gov

Abstract

The idle computers on a local area, campus area, or even wide area network represent a significant computational resource—one that is, however, also unreliable, heterogeneous, and opportunistic. We describe an algorithm that allows branch-and-bound problems to be solved in such environments. In designing this algorithm, we faced two challenges: (1) scalability, to effectively exploit the variably sized pools of resources available, and (2) fault tolerance, to ensure the reliability of services. We achieve scalability through a fully decentralized algorithm, in which the dynamically available resources are managed through a membership protocol. We guarantee fault tolerance in the sense that the loss of up to all but one resource will not affect the quality of the solution. For propagating information reliably, we use epidemic communication for both the membership protocol and the fault-tolerance mechanism. We have developed a simulation framework that allows us to evaluate design alternatives. Results obtained in this framework suggest that our techniques can execute scalably and reliably.

1. Introduction

For solving new, more difficult search problems, scientists need better search heuristics and/or more powerful resources. The need for hundreds or even thousands of processors is justified in the case of branch-and-bound search algorithms by problems that could not be solved after months of execution on tens of processors [7].

Rarely, however, are thousands of processors assembled in a single location and available for a single problem. Thus, techniques are needed that would allow us to aggregate processors at many different Internet-connected locations. These processors are likely often to be required for other purposes; hence their availability will be episodic, and any algorithm designed to take advantage of these resources must be opportunistic. Furthermore, the Internet environment is likely to be unreliable and heterogeneous.

Various groups have demonstrated the feasibility of using Internet-connected computers for solving embarrassingly parallel problems [24, 19]. In our work, we investigate the feasibility of applying Internet-connected resources to more tightly coupled problems, in which a centralized scheme is not computationally efficient. Our approach is to develop specialized algorithms that incorporate scalability and reliability mechanisms.

For providing reliable services over unreliable architectures, researchers usually choose one of the following approaches: (1) embed fault-tolerance mechanisms within the middleware software layer, as in ISIS [2] or CORBA Transaction Service, or as in systems like Condor [22, 30] or Legion [21]; or (2) embed fault-tolerance mechanisms within algorithms. The former approach is more general. Successful results in this domain guarantee communication and hardware reliability to a large number of applications. But its generality imposes problems that sometimes turn out to be unsolvable [11, 3] or very expensive. The latter alternative is applicable to specific problem classes and is therefore less general. But exploiting the characteristics of a class of problems may ease the design of fault-tolerance mechanisms, yielding simpler and more efficient algorithms. Note that middleware can still be of assistance in this case, by providing appropriate fault-detection services [25].

In order to avoid the high costs of general approaches in achieving fault tolerance, in our work we focus on problem-specific fault-tolerance mechanism. Specifically, we propose a fault-tolerant, totally distributed branch-and-bound algorithm designed for unreliable architectures, with a dynamically variable number of resources. The novelty of our work is the fully-decentralized fault-tolerance mechanism that uses a tree-based encoding of the branch-and-bound subproblems. The description of the branch-and-bound problem (Section 2) and the target architecture (Section 4) provide the motivation for our work. We describe our branch-and-bound algorithm in Section 5, focusing particularly on the fault-tolerance mechanism. Because of lack of space, related work (Section 3) refers only to fault-tolerance techniques embedded in tree-based, distributed, asynchronous algorithms. For testing our solution, we have developed a simulation framework, which is presented in Section 6, along with the results obtained. We conclude with a discussion of what we learned from trying to solve this problem and how we intend to continue this work.

2. Branch and Bound

The search for optimal solutions is one of the most important searching problems. Since exhaustive search is often impracticable in NP-hard problems, heuristics are employed to improve search performance. Branch-and-bound (which we shall hereafter refer to as B&B) is an intelligent search method often used for optimization problems. It uses a successive decomposition of the original problem into smaller disjoint subproblems, while reducing (*pruning*) the search space by recognizing unpromising problems before starting to solve them.

A sequential B&B algorithm consists of a sequence of iterations in which four basic operators are applied over a list of problems, called a *pool of active problems*:

a. **Decompose**. Splits a problem into a set of new subproblems. A problem that cannot be split (either because it has no solution or because a solution is found) is *fathomed*. A problem decomposed into new subproblems is *branched*.

b. **Bound**. Computes a bound value $l(v)$ on the optimal solution of subproblem v. This bound value will be used by *Select* and *Eliminate* operations.

c. **Select**. Selects which problem to branch from next, as a function of some heuristic priority function. Selection may depend on bound values, such as in the *best-first* selection rule, or not, as in the case of *depth-first* or *breadth-first* rules.

d. **Eliminate**. Eliminates problems that cannot lead to an optimal solution of the original problem (i.e., problems for which $l(v) \geq U$, where U is the *best known solution*).

Successive decomposition operations create a tree of problems rooted in the original problem. The value of the best solution found thus far is used to recognize the unpromising problems and *prune* the tree. If the bound value of the current problem is not better than the best-known solution, then the problem is eliminated. Otherwise, it is stored into the pool of active problems. The best-known solution is updated when a better feasible solution is found. The leaves of the tree are infeasible problems, or pruned problems, or problems that lead to locally optimal solutions. The size and shape of the tree strongly depends on the quality of the heuristic function for the selection rule.

In B&B algorithms, parallelism can be achieved in different ways [14]. We consider the most general approach, in which the B&B tree is built in parallel by performing operations on different subproblems simultaneously.

Three design choices most influence the performance of parallel B&B algorithms: the choice of a synchronous or an asynchronous algorithm, the *work sharing* mechanism, and the *information sharing* mechanism. Synchronous vs. asynchronous design defines what processes do upon completion of a work unit—they wait for each other (in the case of synchronous algorithms) or not (in asynchronous algorithms). Work sharing is the method used to assign work to processes in order to efficiently exploit available parallelism. Information sharing refers to the methods used to publish and update the best-known solution. Using an up-to-date best-known solution improves the efficiency of the selection and elimination rule and hence has an important effect on the size of the search space.

3. Related Work

Many investigations of parallel B&B for distributed-memory systems have adopted a centralized approach in which a single manager maintains the tree and hands out tasks to workers [14, 26]. While clearly not scalable, this approach simplifies the management of information and multiple processes. Scalability can be improved through a hierarchical organization of processes or by varying the size of work units, but the central manager remains an obstacle to both scalability and fault tolerance. Reliability can be achieved through checkpointing, but this approach assumes that there exists at least one reliable process/machine, able to manage the failure recovery process.

Because of the highly variable number of resources in the architecture we consider, we need more flexibility than that offered by the centralized design. Hence we chose a fully decentralized design.

The only fully decentralized, fault-tolerant B&B algorithm for distributed-memory architectures is DIB (Distributed Implementation of Backtracking) [10]. DIB was designed for a wide range of tree-based applications, such as recursive backtrack, branch-and-bound, and alpha-beta pruning. It is a distributed, asynchronous algorithm that uses a dynamic load-balancing technique. Its failure recovery mechanism is based on keeping track of which machine is responsible for each unsolved problem. Each machine memorizes the problems for which it is responsible, as well as the machines to which it sent problems or from which it received problems. The completion of a problem is reported to the machine the problem came from. Hence, each machine can determine whether the work for which it is responsible is still unsolved, and can redo that work in the case of failure.

4. Target Architecture

The target architecture for our algorithm is a collection of Internet-connected computers. The distinctive characteristics of this environment, when compared with a conventional parallel computer, are as follows:

Scale. The number of resources available can potentially be much larger than on a conventional parallel computer.

Dynamic availability. The quantity of resources available may vary over time, as may the amount of computation delivered by a single resource.

Unreliability. Resources may become unreachable without notice because of system or network failures.

Communication characteristics. Latencies may be high, variable, and unpredictable; bandwidth may be low, variable, and unpredictable. Connectivity (as measured, for example, by bisection bandwidth) may be particularly low.

Heterogeneity. Resources may have varying physical characteristics (for example, amount of memory, speed).

Lack of centralized control. There is no central authority for quality control or operational management.

The failure model we consider is Crash [4, 23], in which a processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed may not be detectable by other processors. We make minimum assumptions about the system:

– There is no bound on message delivery time (asynchronous environment).

– Messages may be lost altogether.

– A network link does not duplicate, corrupt, or spontaneously create messages.

– The clock rate on each host is close to accurate (we do not assume that the clocks are synchronized). This condition is assumed in many works in the fault-tolerance domain and it does not represent a *practical* restriction [5].

5. The Algorithm

We propose a fully decentralized, asynchronous, fault-tolerant parallel B&B algorithm suited for the environment described above. Asynchrony is required by the heterogeneity of the architecture and allowed by the B&B problem [14]. Each process maintains its local pool of problems to be solved. When the local pool is empty, the process sends work requests to other processes. A process that receives a work request and has enough problems in its pool removes some of those problems and sends them to the requester. This on-demand dynamic load-balancing scheme was chosen to reduce unnecessary communication. The fully decentralized scheme was preferred for better scalability and for greater reliability. The information sharing issue is solved by circulating the best-known solution among processes, embedded in the most frequently sent messages. Processes update the local value for the best-known solution every time they receive it, and use it when the next decision is to be made.

For adapting this rather conventional B&B algorithm to the environment described above, we extend it with (1) a group membership protocol to allow dynamic variation in the number and components of resources and (2) a fault-tolerance mechanism. The novelty of this paper is the decentralized fault-tolerance mechanism that relies on a tree-based encoding of the B&B subproblems. This strategy for problem encoding also offers a simple mechanism for termination detection, described in Section 5.4. A brief description of the epidemic communication mechanism (Section 5.1) will help in understanding how the group membership protocol (Section 5.2) and fault-tolerance mechanism (Section 5.3) function. A comparison with DIB, the decentralized B&B algorithm mentioned in Section 3, concludes this section.

5.1. Epidemic Communication for Group Membership and Fault Tolerance

Epidemic communication [1] allows temporary inconsistencies in shared data in exchange for low-overhead implementation. More specifically, information changes are spread gradually throughout the processes, without the overhead and communication costs typically used to achieve a high degree of consistency.

Both our group membership and fault-tolerance mechanisms use epidemic communication. Since these mechanisms do not require data consistency, epidemic communication is a convenient algorithm for spreading information. However, epidemic communication guarantees that eventually consistency is achieved; that is, all processes will eventually see the same data when no more new information is brought into the system, independent of system failures

[9, 17]. This observation is exploited for termination detection.

The epidemic algorithms used are variants of the *rumor-mongering* algorithm (analyzed in [6]): when a site receives a new update (*rumor*), it becomes "infectious" and is willing to share—it repeatedly chooses another member, to which it sends the rumor. Upon receipt of a rumor, a member updates its local information and sends its own version after some time interval. In the membership protocol, the rumor received is sent farther, without being processed. In the fault-tolerance mechanism, the rumor is stored for local processing, may be processed locally, and is spread infrequently.

5.2. Group Membership Protocol

The group membership protocol is used for collecting and updating information about which resources participate in the computation at any given time. The impossibility of guaranteeing consistent views of group membership in asynchronous, unreliable systems was proven in [3]. Even in reliable systems, membership protocols in asynchronous systems are expensive, requiring several phases for consistency.

A group is defined as a set of members. It is initialized when the first member enters the group and ceases to exist when the last member leaves. A process joins a group by finding one or more members of the group and leaves it either by leaving or by failing. We assume the existence of a fault-tolerant method by which processes can find other processes, such as broadcasting (when applicable), known addresses of gossip servers (described below), or a location service. For the moment, we assume that gossip servers exist.

Our membership protocol is inspired by the failure-detection mechanism based on epidemic communication presented in [29]. Other membership protocols based on epidemic communication [15] are more elaborate and introduce constraints or costs that are not justified in our case.

The membership protocol works as follows: when a new computer joins the group of resources, it sends its address to some known gossip servers. The gossip servers act as any other member of the group, except that at least one of them is guaranteed to be active at any given moment during the computation. This is a loose fault-tolerance constraint, easily achievable in the absence of network partitioning problems by increasing the number of gossip servers in the system. The main task of these servers is to propagate information about the newly arrived members.

Each member process maintains a view of group membership. The view defines a set of processes that the member believes are part of the group at any given time. In addition, it contains specific information designed to log the members' activity by keeping track of when it last heard of each (known) member, directly from it or through the gossip system. The parameters involved in this mechanism (for example, the frequency of gossiping and the timeout period used to deduce failure of a passive member) are chosen to keep communication and the probability of false membership information under some threshold values [29].

Unlike most of the existent group protocols for asynchronous systems, whose primary goal is better membership view consistency [28, 2, 20], the goal of our membership protocol is scalability in unreliable systems. Among the advantages of using this membership protocol are (1) scalability in network load with the size of the group, (2) tolerance to a small percentage of message loss or failed members, and (3) scalability in membership view accuracy with the number of members.

5.3. Fault-Tolerance Mechanism

For B&B algorithms, the loss of a subproblem is unacceptable when the accuracy of the solution is important.

Our proposed fault-tolerance mechanism does not attempt to detect failures of computers and restore their data, but rather focuses on detecting missing results. Given that the B&B tree of problems is dynamic, how is it possible to know the set of existing problems, so that, knowing the problems completed, one can infer the set of not-completed problems?

Our solution exploits the fact that the subproblems dynamically generated by the B&B algorithm are nodes of a tree. Each node can be uniquely represented by its position in the tree. If we encode the position of the nodes in the tree, we obtain a unique code for each subproblem. Furthermore, given a set of nodes of the tree, we can easily find its complement, that is, the list of nodes of the tree that are not in the given set.

5.3.1. Problem Representation. Without loss of generality, we assume that the branching factor for the search tree is 2 and that each branch is a decision on a condition variable. Therefore, a subproblem is entirely described by a sequence of pairs $\langle x_i, value\rangle$ where x_i is a condition variable and $value$ is 0 or 1, indicating the left or the right branch, respectively. We need to include condition variables in the subproblem encoding because the order in which condition variables are considered may vary over the tree. For example, the left subtree of a node that branches upon x_k may consider x_i first and therefore will generate the subproblems $(\langle x_k, 1\rangle, \langle x_i, 0\rangle)$ and $(\langle x_k, 1\rangle, \langle x_i, 1\rangle)$, whereas the right subtree may branch upon x_j first, producing the subproblems $(\langle x_k, 0\rangle, \langle x_j, 0\rangle)$ and $(\langle x_k, 0\rangle, \langle x_j, 1\rangle)$.

Each pair $\langle x_i, value\rangle$ introduces and assigns a condition to a new variable. That is what makes the codes (subprob-

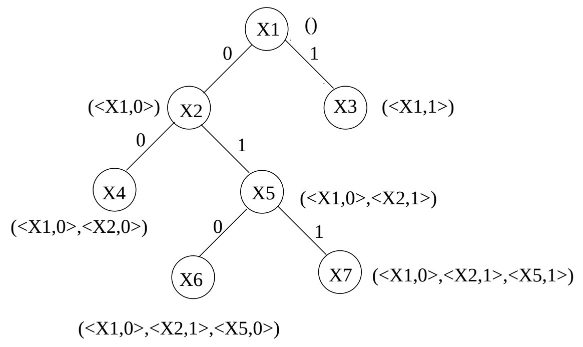

Figure 1. Problem representation

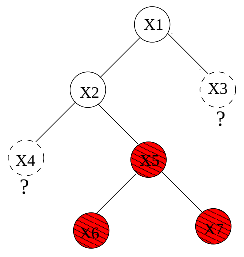

Figure 2. Completed, unsolved, and solved problems: shadowed nodes represent completed problems, dashed nodes represent unsolved problems (i.e., problems that are still in the active pool), and plain nodes represent solved but uncompleted problems.

lems) self-contained: the code (along with the initial data, which is provided by a gossip server when a process joins the computation) is enough to initiate a problem on any processor.

5.3.2. Mechanism Description. Our failure-recovery mechanism allows each process to detect missing problems independently, based on local information about completed problems.

We consider a subproblem *solved* after the branching operation has been performed on it. *Solved* subproblems are not necessarily *completed*: we consider a subproblem to be *completed* if it is solved and either it is a leaf or both its children are completed (see Figure 2).

Every process maintains a list of new locally completed subproblems and a table of the completed problems it knows about. When a problem is completed, it is included in the local list. When c problems (codes) are in the list or the list has not been updated for a long time, the list is sent to m of the other members as a *work report* message. When a member receives a work report, it stores the report in its table. Occasionally, in order to inform new members of the current state of the execution and to increase the degree of consistency, a member sends its table of completed problems to a randomly chosen member.

The size and the number of the problem codes vary with the shape and number of nodes of the B&B tree. The deeper the node in the tree, the larger the size of its code; the more nodes in the tree, the larger the number of codes. Since the completion of a parent node implies the completion of its children, communication costs can be reduced by compressing work report messages, via the recursive replacement of pairs of sibling codes with the code of their parent, and the deletion of codes whose ancestors are also in the list. Simulations performed on real B&B trees confirmed that the compression rate is better when processors are sufficiently loaded: the taller the subtree completed locally, the larger the number of codes that do not need to be sent.

Failure recovery is achieved as follows. When a member runs out of work and an attempt to get work through the load-balancing mechanism fails, it chooses an uncompleted problem (by complementing the code of a solved problem whose sibling is not solved) and solves it. The mechanism "repairs" system failures due to, for example, a computer that failed before sending work reports or work reports that were lost before reaching any machine. Note that this mechanism also works in the case of temporary network partitions.

This simple, fully distributed mechanism can lead to redundant work in two situations: (a) the lag in updating information can lead to faulty presumptions on failure; and (b) the lack of coordination among processors permits multiple members to work on the same problem. The former case can be fixed easily by interrupting the redundant work when information is updated. The costs of the latter situation can be reduced by employing more sophisticated methods for choosing work, such as using the location of the last problem completed locally. Notice, however, that redundant computation may be inevitable.

If information about completed problems is spread uniformly, then the loss of a percentage of members may not lead to information loss: if the information about the problems reported to be completed still exists in the system, they will not have to be redone.

5.4. Almost Implicit Termination Detection

The problem encoding used for implementing the fault-tolerance mechanism also has the advantage of implicitly solving the termination detection problem. When successive code compressions of local lists and tables lead to the code of the root problem, termination is detected. Since none of the communication mechanisms used guarantees data consistency, it is possible that some members do not have enough information to detect termination. That is why, before termination, each member that detected the termination will have to send one more work report, that is, the code of the root problem, to all members from its local membership list.

5.5. Comparison with DIB

Both DIB and the algorithm we propose are decentralized and fault-tolerant algorithms that work on a dynamic, tree-like search space. Both algorithms implement low-cost, simple fault-tolerance protocols for the price of potentially redundant work. However, the two algorithms have different failure-recovery mechanisms and react differently in the case of failure.

DIB uses a hierarchical structure for failure detection and recovery that imposes the need for a reliable or duplicated node for the root of this hierarchy. Moreover, the failure of a node affects not only the problems solved locally and not reported as solved yet, but also the problems given to other nodes, whose completion cannot be reported (and therefore considered) anymore.

In our algorithm, all processes are equally responsible for the behavior of the system in case of failure. Our simulation studies confirm that the failure of all processes but one still allows the problem to be correctly solved. The mechanism is also reliable in the case of faulty network links or temporary network partitions.

However, the homogeneity involved in our algorithm has a communication cost: information about the completion of a problem is eventually spread to all processes, directly (by reporting the code of the problem) or indirectly (by reporting the completion of one of its ancestors).

Performance comparisons of DIB and our algorithm are of limited interest for two reasons. Because DIB was designed for a wide range of applications, such as recursive backtrack, alpha-beta search and branch-and-bound, its speedup is “excellent for exhaustive traversal and quite good for branch-and-bound” [10]. Furthermore, speedup results are given for maximum 16 processors, while we are interested in many more resources.

6. Experimental Studies

We use simulations rather than a real implementation to evaluate our algorithm, as the use of simulation techniques provides great flexibility in testing a wide range of B&B strategies in a variety of Internet-like environments.

6.1. Experimental Goals

The goals of our experimental work are as follows: (1) to verify reliability and evaluate the overall performance of the algorithm, focusing on the costs introduced by the fault-tolerance mechanism; and (2) to evaluate scalability for different problem classes and environments. Our work to date has focused primarily on the first of these two issues.

We studied algorithm reliability by testing various failure scenarios. The costs introduced by our fault-tolerance mechanism are communication costs, storage space, tree contraction time, and redundant work. Because we avoid centralized control by spreading information throughout the system, communication costs may be significant. Redundant work may increase when communication conditions are poor (messages are delayed or lost) or when work load is low. Storage space may become a serious concern for large problems because the algorithm permits (and benefits from) the replication of data. However, the results we obtained encourage us to continue our research in this direction.

6.2. Simulation Framework

We used Parsec [27] to develop our simulation system. Parsec is a C-based simulation language for sequential and parallel execution of discrete-event simulation models. Processes are modeled by objects; interactions among objects are modeled by time stamped message exchanges.

Our simulation system incorporates a detailed representation of load balancing, failure recovery, and termination detection mechanisms. We do not include yet the membership protocol: hence, the pool of resources is predetermined and varies only with failures. Each process, after it has solved a B&B subproblem, checks to see whether any messages are pending. If it received a work request, it satisfies the request if there are enough problems in its active pool. If it received a work report, it merges that report with its local information on completed problems and contracts the result.

The simulation was configured so that it could be driven either by *real* (precomputed) B&B trees or by random trees. For real problems, we tested our algorithm on a set of *basic trees* that we obtained from an instrumented B&B code. Basic trees are trees generated by executing a branch-and-bound algorithm without eliminating the unpromising nodes.

For each node in the tree, we have the following information: (1) the node identifier, (2) its bound value, (3) the time needed for computing the bound value and expanding the node or determining infeasibility, and (4) a value specifying whether the bound value is a feasible solution. The bound values are used for pruning the test tree and obtaining the B&B tree, and for computing the optimal solution. The time value is used for simulating the execution time needed for the bounding operation. Notice that the time values determine the granularity of the subproblems. During our experiments, we tuned this granularity by multiplying all time values by a constant factor, and we studied how granularity affects the overall performance of the B&B algorithm.

Running simulations on basic trees leaves enough room for generating different B&B trees, depending on communication characteristics (for example, up-to-date information about the best-known solution influences pruning decision) and on the number of processors (because the number of nodes expanded may vary with the number of processors). Note that the basic branch-and-bound operation *decompose* is recorded within the basic tree structure.

Because the amount of communication and storage space depends on the shape and the size of the tree, testing trees resulted from solving real problems provides better accuracy. However, creation of basic trees is computationally infeasible for anything but small problems. But for testing reliability, and later scalability, the number of nodes is the only important feature of the test tree. Therefore, we enriched our set of test trees with randomly created trees of various sizes and tested them without eliminating the unpromising nodes.

6.3. Results

Our simulator measured execution time, communication costs, and storage space. We tested the algorithm on relatively small problems (up to tens of thousands of nodes expanded), with no optimization efforts: work reports are sent to randomly chosen resources, without eliminating redundant messages. When out of work, resources ask randomly chosen resources for work, without using previous experience to increase performance.

6.3.1. Algorithm Performance. For a very small problem (approximately 3500 nodes expanded and average granularity of 0.01 seconds per node) the overhead introduced by the algorithm reaches 36% for 8 processors. This is determined by three factors: (1) the relatively high communication costs considered ($1.5 + 0.005 \times L$ milliseconds for messages of size L bytes); (2) the cost of the dynamic load balancing mechanism for a network of workstations [16]; and (3) the small granularity of the subproblems. We shall see that for a larger problem (Table 1) the overhead is much lower (15.58% of the total execution for 100 processors, from which 13.67% are load balancing costs, 0.78% communication time and 1.13% list contraction time). Furthermore, this overhead can be controlled by tuning various execution parameters. For example, less frequent termination verification leads to lower list contraction costs but may increase idle time. Sending work reports more rarely may decrease communication time and list contraction costs but may increase termination detection time, because of lack of information. If the failure recovery mechanism is activated (decides that a problem was lost and recreates it) less often, the overhead introduced (list contraction and redundant work costs) is lower, but recovery in case of failure is also slower.

The tests we performed on larger problems (total uniprocessor execution time of around 75 hours) show that communication and storage space costs remain negligible (Table 1). We find that good performance is achieved on up to 100 processors. These preliminary results encourage us to continue evaluating our algorithm on larger problems, with larger number of resources.

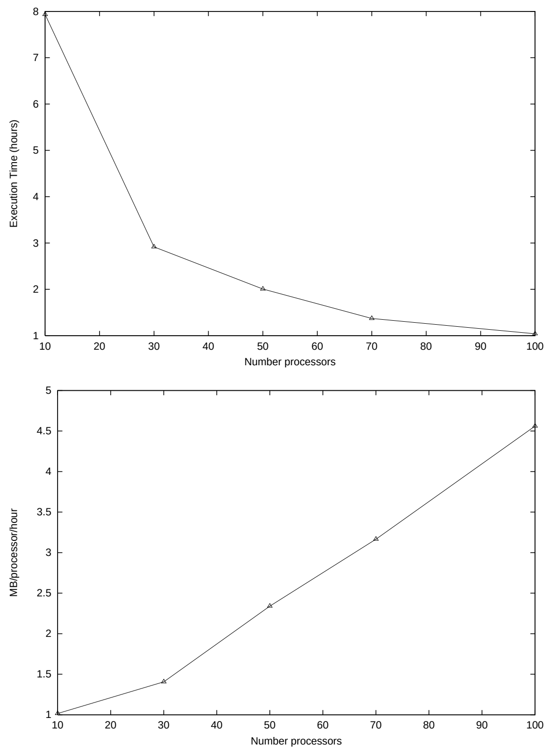

Figure 3. Execution time and communication in the problem from Table 1.

Communication per processor increases with the number of processors because the number of work reports sent per processor increases: since the work load is lower, and therefore processes are idle longer periods of time, they suspect termination and send more work reports. Storage space is measured for the entire system. The results obtained—43 MB storage space for 100 machines—are promising.

A normal trend would be that the amount of time spent on list contraction increases with the number of processors, since the number of messages circulated within the system increases and the receiving of a work report message requires a list contraction procedure. But because this depends on how the subproblems are assigned on processors, a lucky configuration may lead to unexpected good results

Table 1. Simulated execution of a real problem (approximately 79,600 nodes expanded). In this problem, average cost per node is 3.47 seconds. Communication costs are modeled as $1.5 + 0.005 \times L$ milliseconds for messages of size L bytes. B&B time and contraction time are respresented as percentage of execution time.

No. Processors	Execution Time (hours)	B&B Time (%)	Contraction Time (%)	Storage Space Total (MB)	Storage Space Redundant (MB)	Communication (MB/hour per processor)
10	7.93	98.11	0.35	0.42	0.16	1.01
30	2.91	90.42	5.20	3.76	1.92	1.40
50	2.00	81.19	11.73	12.65	6.43	2.34
70	1.37	87.32	2.33	19.81	10.13	3.16
100	1.04	84.40	1.13	43.06	21.88	4.56

(as for 100 processors, Table 1).

The amount of redundant work is another interesting measure of our algorithm performance that remains to be evaluated. However, this amount can be reduced by tuning parameters (for example, how soon failure is suspected after a machine unsuccessfully tries to get work) or by designing more sophisticated methods for picking up unsolved problems.

When varying problem granularity (by multiplying the time needed to solve a problem with some constant values), we observed the following (not unexpected) behavior: The number of nodes expanded may vary, because the information of the best-known solution is computed at different moments. Load balancing costs are lower when granularity is coarser. Communication increases unnecessarily because work reports are sent at fixed time intervals. This last observation taught us that for scalability, we need to design an adaptive mechanism for deciding how often work reports should be sent, based on information collected at runtime: for example, information about execution time per subproblem and frequency of messages received.

6.3.2. Fault Tolerance. Because our termination detection mechanism operates by detecting that all expanded problems have been completed, it is straightforward to verify that our fault-tolerance algorithm is working correctly—we simply verify that termination is detected. For visualizing the behavior of the algorithm, we used Jumpshot, a graphical visualization tool for `clog` log file format. We used the MPE library developed by the MPICH team at Argonne National Laboratory for logging the execution profile.

Figures 4 and 5 are snapshots of the execution of the algorithm on a very small problem. Figure 4 shows the behavior of the algorithm in the absence of failures. The same problem is presented in Figure 5, where two of the three processors fail at about 85% of the execution time. The only processor available after this moment is able to solve the problem and terminate.

7. Conclusions and Future Work

We presented a failure-recovery mechanism suited for a tree-like problem space. This mechanism and a low-cost group membership protocol are the ingredients that transform a rather conventional parallel branch-and-bound algorithm into a scalable, reliable, more powerful algorithm, able to exploit the computational power of hundreds of Internet-connected resources. Scalability is achieved through a fully distributed design. The algorithm is fault tolerant under our assumptions and can execute and terminate correctly even if only a single resource remains available.

We solved the difficult problems of fault tolerance and termination detection in distributed environments by exploiting problem-specific features, specifically the tree structure of the problem space. While the mechanism we propose is not applicable to all distributed computations, we believe that a large class of problems can benefit from it.

We have used simulation studies to explore the behavior of our algorithm. Initial results on relatively small problems and up to 100 processors are promising: performance is good despite the lack of optimization. Communication costs are reasonable, storage space costs are negligible. However, we need results on a much larger number of processors. We plan to introduce the group membership protocol into our simulations and to test the algorithm under various network conditions. An interesting issue to study is how the network characteristics influence the performance of the algorithm in general and the costs introduced by the failure-recovery mechanism in particular. Also, in order to accurately analyze scalability issues, we plan to design a flexible scheme for adapting parameters to runtime informations, such as total execution time and execution time per problem.

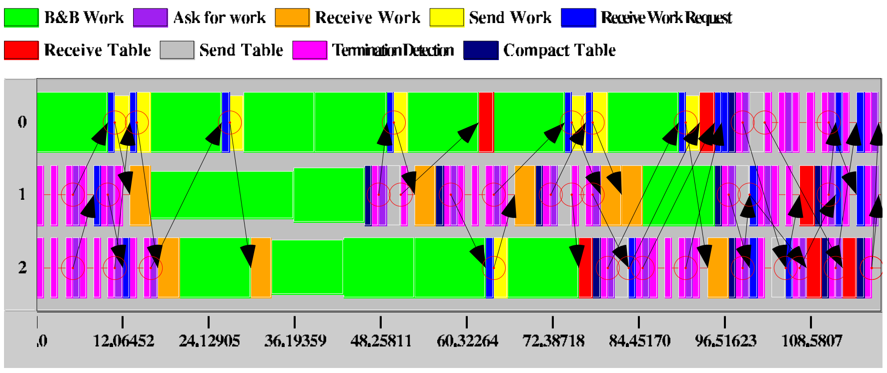

Figure 4. Solving a very small problem on 3 processors. The X axis represents time in ms.

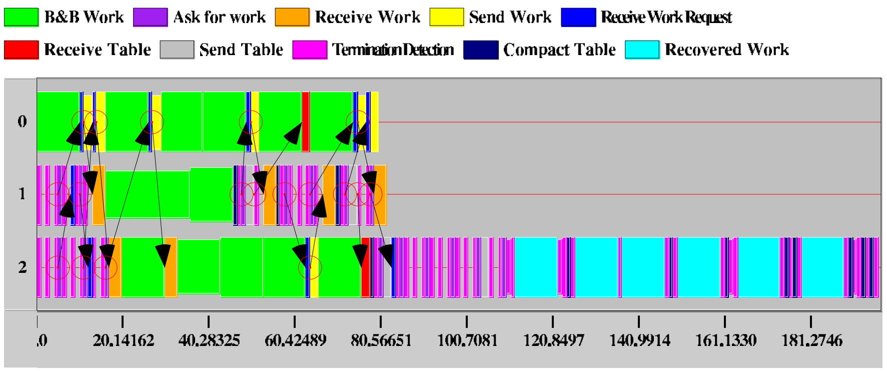

Figure 5. The same problem as in Figure 4: two processors crash about the same time; the third processor recovers the lost work.

Acknowledgments

We are grateful to Jean-Pierre Goux, Qun Chen and Jeff Linderoth, both for providing us data for our experiments and for helpful discussions concerning this research.

This work was supported by the National Science Foundation grant CDA-9726385.

References

[1] N. Alon, A. Barak, and U. Mander. On disseminating information reliably without broadcasting. In *7th International Conference on Distributed Computing Systems ICDCS97*, Berlin, Sept. 1987. IEEE Press.

[2] K. P. Birman. ISIS: A system for fault-tolerant distributed computing. Technical Report TR86-744, Cornell University, Computer Science Department, Apr. 1986.

[3] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, New York, USA, May 1996. ACM.

[4] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, Feb. 1991.

[5] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[6] A. Demers, D. Greene, H. C., W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *ACM Operating Systems Review, SIGOPS*, 22(1):8–32, Jan. 1988.

[7] J. Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM–5. *SIAM Journal on Optimization*, 4:794–814, 1994.

[8] J. Eckstein. How much communication does parallel branch and bound need? *INFORMS Journal on Computing*, 9:15–29, 1997.

[9] A. Farley. Minimum-time broadcast networks. *Networks*, 10:59–70, 1980.

[10] R. Finkel and U. Manber. DIB — A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, Apr. 1987.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[12] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann, 1999.

[13] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, Mar. 1999.

[14] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[15] R. A. Golding and K. Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, University of California, Santa Cruz, Jack Baskin School of Engineering, Mar. 1992.

[16] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, July 1994.

[17] C. Laforest. Broadcast and gossip in line-communication mode. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 80:161–176, 1997.

[18] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report CS-TR-1997-1346, University of Wisconsin, Madison, Apr. 1997.

[19] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A hunter of idle workstations. In *Proc. 8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, San Jose, Calif., June 1988.

[20] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *11th International Conference on Distributed Computing Systems*, pages 480–489, Washington, D.C., USA, May 1991. IEEE Computer Society Press.

[21] A. Nguyen-Tuong and A. S. Grimshaw. Using reflection for incorporating fault-tolerance techniques into distributed applications. Technical Report CS-98-34, Department of Computer Science, University of Virginia, Nov. 5 1998.

[22] J. Pruyne and M. Livny. Managing checkpoints for parallel programs. In *Workshop on Job Scheduling Strategies for Parallel Processing IPPS '96*, pages 268–278, 1996.

[23] F. B. Schneider. What good are models and what models are good? In S. Mullender, editor, *Distributed Systems*, chapter 2, pages 17–26. Addison-Wesley, second edition, 1993.

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

[25] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 268–278. IEEE, 1998.

[26] H. Trienekens and A. de Bruin. Towards a taxonomy of parallel branch-and-bound. Technical Report EUR-CS-92-01, Erasmus University, Rotterdam, 1992.

[27] UCLA Parallel Computing Laboratory. *Parsec. http://may.cs.ucla.edu/projects/parsec*.

[28] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR94-1442, Cornell University, Computer Science Department, Aug. 1994.

[29] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, Cornell University, Computer Science, May 1998.

[30] V. Zandy, B. Miller, and M. Livny. Process hijacking. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 177–184. IEEE, 1999.