Parallel Branch and Bound Algorithms on Hypercube Multiprocessors*

Tarek S. Abdelrahman and Trevor N. Mudge

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109

Abstract

Branch and Bound (BB) algorithms are a generalization of many search algorithms used in Artificial Intelligence and Operations Research. This paper presents our work on implementing BB algorithms on hypercube multiprocessors. The 0–1 integer linear programming (ILP) problem is taken as an example because it can be implemented to capture the essence of BB search algorithms without too many distracting problem specific details. A BB algorithm for the 0–1 ILP problem is discussed. Two parallel implementations of the algorithm on hypercube multiprocessors are presented. The two implementations demonstrate some of the tradeoffs involved in implementing these algorithms on multiprocessors with no shared memory, such as hypercubes. Experimental results from the NCUBE/six show the performance of the two implementations of the algorithm. Future research work is discussed.

1 Introduction

Hypercube multiprocessors have been used successfully for a wide range of applications in science and engineering. These applications can be characterized to a large extent by:

1. The applications tend to have uniform data sets that are constant in size and homogeneous in nature.
2. There is a large degree of inherent parallelism in these applications.
3. The work load generated by an application is uniform across the processors; usually as a consequence of dividing the data set uniformly among the processors.

In this paper, we consider the class of BB algorithms. These algorithms do not exhibit the above characteristics and hence provide a challenge for parallel processors that, like hypercubes, do not have shared memory. It is not clear how much parallelism exists in a BB algorithm. The algorithm generates an irregular data set (the BB tree) dynamically. This implies that the division of work among the processor has to be done dynamically and that a load balancing mechanism must be employed. The BB algorithm accumulates knowledge about the problem it is solving during its execution and uses that knowledge to improve its efficiency. In a distributed memory environment, maintaining that knowledge involves a number of interesting tradeoffs.

The remainder of this paper is organized as follows. The general formulation of the BB algorithm is described in Sec. 2. The 0–1 ILP problem and the specific BB algorithm used to solve it are described in Sec. 3. Two parallel implementations of that algorithm on a hypercube multiprocessor are described in Sec. 4. The results obtained from our experimental setup on an NCUBE/six [NCUB85] system are given in Sec. 5. Some concluding remarks and future research directions are given in Sec. 6.

2 The Branch and Bound Algorithm

There is a large class of problems in the fields of Operations Research and Artificial Intelligence for which there exists no "direct" methods of solution or no efficient ones. Finding a solution involves a search through the problem space. Unguided search, however, can easily become inefficient since many of these problems are at least NP-complete. Several techniques have been developed to guide the search and improve its average efficiency. The most general of these techniques is the Branch and Bound (BB) algorithm [LaWo66].

Many aspects of BB algorithms have been studied in the literature. In [Ibar76a,Ibar76b,Ibar77,Ibar78] many of the theoretical properties of BB algorithms have been developed. BB algorithms have been applied to solve a wide variety of problems. Examples include: traveling salesman [LMSK63],

*This work was supported in part by the Robot Systems Division, University of Michigan and by the Materials Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, Ohio 45433-6503.

integer programming [GeMa72], knapsack [InKo77], and many others. Also, BB algorithms have been recognized to be a generalization of many of the heuristic search algorithms in Artificial Intelligence such as A*, AO* and alpha-beta [KuKa83].

The BB algorithm is an intelligent structured search of the problem space. The algorithm can be best described as a partitioning algorithm which conducts its search for a solution by partitioning the problem space into subspaces of decreasing size until the desired solution is found or its non-existence is determined.

The BB algorithm, in its most abstract form, consists of two processes: a branching process and a bounding process. The branching process partitions the problem space, or subspaces of it, into smaller size subspaces. The branching process always partitions first the subspace which is most likely to contain the desired solution. The branching process continues until the subspace is small enough to be searched exhaustively to determine if it contains the desired solution.

The bounding process of the algorithm acts to reduce the number of subspaces partitioned by the branching process. A subspace is examined by the bounding process before it is partitioned. If it is proven that the subspace does not contain the desired solution, the subspace is *pruned* or eliminated from further consideration by the branching process.

The combined action of the branching and bounding processes reduces the extent of the search and improves the search efficiency of the BB algorithm. The branching process guides the search towards solutions by partitioning subspaces that are more likely to contain these solutions before subspaces that are less likely to contain them. The bounding process helps by eliminating subspaces that cannot lead to these solutions *without* actually partitioning these subspaces.

The branching process applied to the problem space of a given problem can be performed by building a search tree, called a *BB tree*, over the problem space of that problem. The root of the tree represents the complete problem space. Children nodes in the tree represent subspaces of the problem space. The branching process proceeds from the root of the tree to the leafs of the tree partitioning subspaces into smaller and smaller subspaces. The leaf nodes represent subspaces that are small enough to be exhaustively searched for solutions.

Subspaces of the problem space represent partial solutions of the problem. Consequently, each node of the BB tree represents one partial solution to the original problem. The branching process proceeds from the root of the tree to the leafs extending partial solutions towards more complete solutions. Each child node represents one possible way of extending its parent's partial solution towards a more complete one.

In most problems it is not practical, if not impossible, to explicitly represent the problem space or subspaces of it. A more practical representation is to use a problem specific data structure which implicitly represents the problem space. This data structure representation is referred to as a *subproblem*. Hence, a subproblem is a representation of a problem subspace or equivalently, a partial solution to the problem. BB algorithms are generally expressed and formulated in terms of subproblems rather than in terms of problem subspaces.

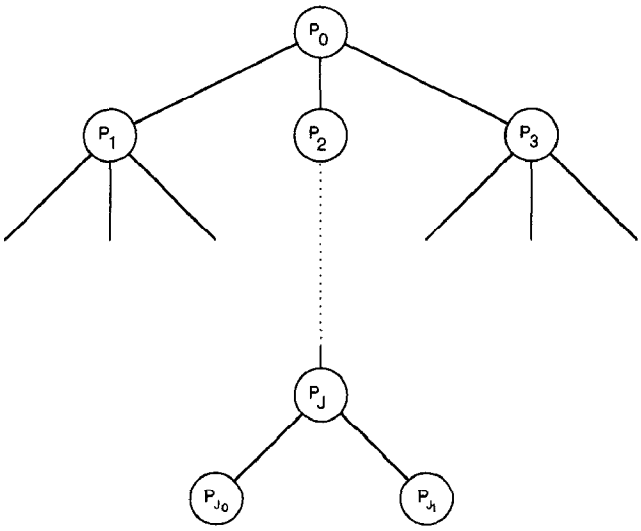

Figure 1: BB tree of a simple example.

The above process of building a BB tree is illustrated in Fig. 1. The figure shows the BB tree for a simple example. The original problem P_0 is at the root of the tree. P_0 is then partitioned into three smaller subproblems P_1, P_2 and P_3. These subproblems are represented as three children of P_0. Each one of the three subproblems is further partitioned into yet smaller subproblems. In general, a subproblem P_j is partitioned into k smaller subproblems $P_{j_1}, \ldots, P_{j_k}$ which are represented as the k children of P_j. The process of partitioning a subproblem into smaller subproblems and adding the new subproblems to the tree is referred to as *expanding* that subproblem. At any point during the search, the set of subproblems that have been generated by the branching process, but have not yet been expanded is referred to as the set of *active subproblems*. The distance from the root of the BB tree to any subproblem, measured in the number of edges, defines the *level* of that subproblem.

A subproblem P_i can be characterized by the value of a cost function f. The function f is defined as the value of the best solution that can be obtained from the subproblem P_i. The value of the function f is not known, however, until the subtree rooted at P_i is completely expanded.

A subproblem P_i can be also characterized by the value of another function g, which is referred to as the *lower bound* function. It is defined as follows:

1. $g(P_i) \leq f(P_i)$ (g is a lower bound estimate of f),
2. $g(P_i) = f(P_i)$ (g is exact when P_i is feasible), and

3. $g(P_{i_j}) \geq g(P_i)$ (lower bounds of descendant nodes never decrease).

That is, the lower bound function is a lower bound estimate of the actual cost function f. In general, g should be much easier to compute than f.

A BB algorithm consists of four major components: a selection procedure, a branching procedure, an elimination procedure, and a termination test procedure.

The selection procedure selects a subproblem from the set of active subproblems. It is based on a selection heuristic function h. The selection procedure always selects the subproblem whose h value is minimum from the set of active subproblems. In other words, the selection procedure determines the order in which the subproblems are selected for expansion. Three heuristics are commonly used. In *best-first*, the heuristic function is the same as the lower bound function. Therefore, subproblems with smaller lower bounds are selected first. In *breadth-first*, the heuristic function is the same as the level of a subproblem in the BB tree. Therefore, subproblems with smaller level numbers are selected first. In *depth-first*, the heuristic function is defined as the negative of the level of a subproblem in the BB tree. Therefore, subproblems that are deeper in the tree are selected first.

It is often convenient to represent the set of active subproblems as an ordered list of subproblems. The order in which the list is maintained is determined by the heuristic function h. In depth-first, the list is maintained in a last-in-last-out order. In breadth-first, the list is maintained in a first-in-first-out order. Finally, in best-first, the list is maintained by increasing lower bound values.

The branching procedure examines the subproblem that has been selected by the selection procedure and uses problem specific methods to break that subproblem into smaller size subproblems.

The elimination procedure examines the newly created subproblems by the branching procedure and deletes the ones that can not lead to better solutions than those already found. To accomplish this, a special subproblem referred to as the *incumbent* is used to store the best feasible solution discovered during the search. A subproblem is deleted if its lower bound is greater than or equal to that of the incumbent.

Finally, the termination test procedure eliminates a new subproblem if that subproblem can not lead to any feasible solutions. Like the branching procedure, problem specific techniques are used to accomplish that task.

The BB algorithm may be formulated as shown below.

1. **Initialization.**
   (a) The set of active subproblems is initialized to contain the original subproblem.
   (b) The incumbent is initialized to ∞.
2. **Selection.**
   (a) The subproblem with the smallest value of the heuristic function h is selected from the set of active subproblems.
   (b) The subproblem is deleted from the set.
3. **Branching.**
   (a) The branching rule is used to generate new smaller subproblems from the one selected in (2). The lower bounds of the new subproblems are calculated.
   (b) Steps 4–7 are performed for each new subproblem generated in 3(a).
4. **Termination Test.**
   (a) The subproblem is evaluated to determine if it can lead to a feasible solution. If not, it is deleted.
5. **Feasibility Test.**
   (a) The subproblem is evaluated to determine if it is a feasible solution. If it is, and its lower bound is smaller than that of the incumbent, it replaces the incumbent. Otherwise, it is deleted.
   (b) If the incumbent is updated in 5(a), then all the subproblems in the set of active subproblems whose lower bounds are greater than or equal to that of the new incumbent are also deleted from the set.
6. **Lower bound test.**
   (a) If the lower bound of a new subproblem is greater than the lower bound of the incumbent, the subproblem is deleted.
7. **Algorithm Termination.**
   (a) If the set of active subproblems is not empty, steps (2)-(6) are repeated. Otherwise, the algorithm terminates, and the optimal solution is the incumbent.

3 The 0–1 ILP Problem

The 0–1 ILP problem is an optimization problem in which it is desired to minimize the value of a linear objective function $f(x_1, x_2, \ldots, x_n)$ subject to a set of constraints. The variables $(x_1, x_2, \ldots, x_n)$, which are referred to as the decision variables, can take only the values 0 or 1. The problem can be more formally stated as follows:

$$\text{Minimize} \quad f = \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad i = 1, 2, \ldots, m$$

$$x_j \in \{0, 1\} \qquad j = 1, 2, \ldots, n$$

It can be assumed, with no loss of generality, that the coefficients c_j, $j = 1, 2, \ldots, n$ are non-negative.

The BB algorithm used to solve the 0–1 ILP problem is known as *implicit enumeration* [WuCo80]. There are n binary variables and the problem could, conceivably, be solved

by enumerating all of the 2^n possible solutions. The bounding process of the BB algorithm implies, however, that many of these 2^n solutions will be discarded without explicitly enumerating them; hence the name "implicit enumeration".

The implicit enumeration algorithm can be described using the following simple terminology. The assignment of a 0 or 1 value to each one of the decision variables gives one of the 2^n possible *solutions*. The assignment of values to some but not all of the decision variables gives a *partial solution*. A partial solution represents a subspace of the solution space or a *subproblem* of the original problem. The decision variables that are assigned values in a partial solution are said to be *fixed*. In contrast, the decision variables with no assigned values are said to be *free*. A *completion* is made by assigning a value of 0 or 1 to one of the free variables.

Since $c_j \geq 0$ for all j, a lower bound f_L on the value of the objective function for any subproblem can be computed by assigning the value of 0 to each free variable. Hence,

$$f_L = \sum_{\substack{fixed \\ variables}} c_j x_j \qquad (1)$$

Furthermore, a constraint can be satisfied if and only if

$$\sum_{\substack{free \\ variables}} max(a_{ij}, 0) \geq b_i - \sum_{\substack{fixed \\ variables}} a_{ij} x_j \qquad i = 1, 2, \ldots, m. \qquad (2)$$

Therefore, it is possible to check the infeasibility of any subproblem by applying equation (2) to the constraints of the problem. Assigning the value of 0 to each free variable in a subproblem makes a special completion that is referred to as the *lower bound completion*. The feasibility of the lower bound completion can be checked using equation (2) which reduces to:

$$\sum_{\substack{fixed \\ variables}} x_j \geq b_i \qquad i = 1, 2, \ldots, m. \qquad (3)$$

The implicit enumeration BB algorithm for the 0–1 ILP problem can be formulated in the following steps:

Step 1 The incumbent, denoted by f_U, is created to contain the best feasible solution found during the search. The lower bound of f_U is initialized to ∞. The initial subproblem, in which all the variables are free, is created. A list of active subproblems is created and the initial subproblem is inserted on it.

Step 2 The subproblem whose lower bound is the smallest among all subproblems on the list of active subproblems is selected.

Step 3 A free variable, x_k, in the selected subproblem is chosen and is used to generate two new subproblems. The first subproblem is generated by making the completion $x_k = 0$. The second is generated by making the completion $x_k = 1$. The variable x_k is now fixed.

Step 4 The lower bound of each new subproblem is calculated using equation (1). The infeasibility of each subproblem is checked using equation (2). The feasibility of the lower bound completion is also checked using equation (3).

Step 5 A subproblem is deleted if any one of the following conditions is true:

a. $f_L \geq f_U$.

b. The subproblem is infeasible.

c. There are no remaining free variables.

d. The lower bound completion is feasible. In this case, the incumbent is replaced by the lower bound completion if $f_L < f_U$, and all subproblems on the list of active subproblems with $f_L \geq f_U$ are deleted.

A subproblem that is not deleted is added to the list of active subproblems.

Step 6 Steps 2–5 are repeated as long as there are subproblems on the list of active subproblems. When the list is empty, the algorithm terminates. The optimal solution is the current incumbent.

The implicit enumeration algorithm illustrates the steps of the general BB formulation of Sec. 2. Step 1 of the algorithm implements the initialization step. Step 2 implements selection. The lower bound of a subproblem is used as the selection heuristic function making the search strategy of the algorithm best–first. Steps 3 and 4 implement branching. Step 5.a implements the lower bound test. Steps 5.b and 5.c implement the termination test. Step 5.d implement the feasibility test. Finally, step 6 implements algorithm termination.

4 The Parallel Algorithms

A number of researchers have studied parallel BB algorithms. In [LaSa83,QuDe85,QuDe86] speedups theoretically achievable by parallel BB are analyzed. In [Moha83], two parallel BB algorithms for the traveling salesman problem for the Cm* are discussed. In [WaLY85] Manip, a multicomputer designed specifically for parallel BB algorithms is described. In [FiMa85] distributed implementation of depth first BB algorithms on a ring machine are considered. In [Quin86] parallel best first BB algorithms for solving the traveling salesman problem on hypercube multiprocessors are considered. In [AnCh86] a parallel implementation of BB algorithms on a hypercube is proposed.

Many of the above researchers have adopted a simplified model for the parallel execution of the BB algorithm. We refer to that model as the *logical model* of the parallel execution of the BB algorithm. The model is depicted in Fig. 2. It consists of

1. A set of N processors (PEs).

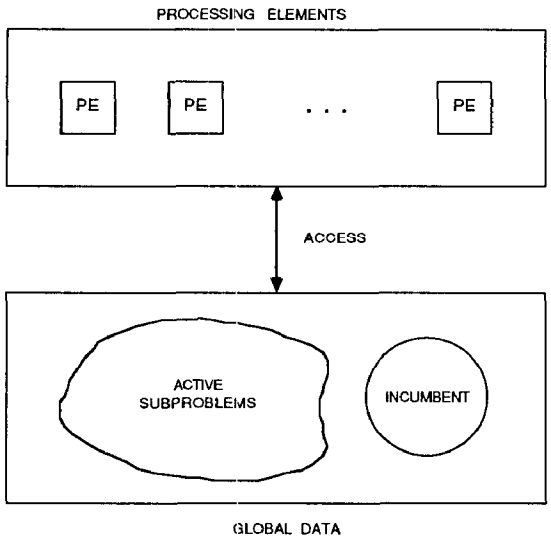

Figure 2: The logical model of execution.

2. Global data which consists of the list of active subproblems and the incumbent. The global data is accessible by all the processors. It is assumed that no overhead is incurred by a processor when it accesses the global data.

3. The processors are synchronized into cycles. Each cycle consists of three steps:

   (a) Each processor selects a subproblem from the N subproblems whose lower bounds are the best among all those on the list of active subproblems.

   (b) Each processor, independently from the other processors, expands its subproblem and performs lower bound, feasibility, and termination tests on the newly generated subproblems.

   (c) The processors insert the newly created subproblems back on the list of active subproblems.

   The processors continue to iterate until the list of active subproblems become empty. The algorithm then terminates and the solution is stored in the incumbent.

While the above parallel model may be adequate for shared memory multiprocessors, it is clear that it is not adequate for distributed memory multiprocessors. In the later case, there is no globally shared memory to facilitate the storage of the global data. Furthermore, in distributed memory multiprocessors, each processor executes it own code which makes synchronizing the processors into cycles difficult.

In the remainder of this section, we consider two parallel implementations of the 0-1 ILP problem BB algorithm on hypercube multiprocessors. The first algorithm, referred to as the Central List (CL) algorithm, is an attempt to directly map the logical model on the hypercube multiprocessor. The second algorithm, referred to as the Distributed List (DL) algorithm, maps the logical model on the hypercube multiprocessor more efficiently and effectively than the CL algorithm.

4.1 The Central list Algorithm

The CL algorithm consists of two major components: a master process and N slave processes. The master process maintains the global data. The slave processes perform the computations necessary for the expansion of subproblems.

The master process operates in iterations in a similar fashion to the logical model. The master process selects N subproblems from the list of active subproblems and assigns one subproblem to each slave process. The N subproblems selected have the best bounds among those subproblems in the list of active subproblems. Each slave process then expands its subproblem and generates children subproblems calculating their lower bounds. Each slave process also performs the lower bound, feasibility and termination tests on the subproblems it generated. The results are then sent back to the master process, which inserts then on the list. The algorithm terminates when the list of active subproblems becomes empty and all the slave processes are idle.

In our implementation, the host processor of the NCUBE/six runs the master process. Each of the processing nodes, runs a slave process. After the master process has selected the subproblems, the host sends subproblems, one to each node. The nodes process the subproblems and send the results back to the host.

The algorithm has the advantage of expanding subproblems whose bounds are best in the global sense. This is advantageous since subproblems that have smaller lower bounds are more likely to lead to solutions than others that have larger lower bounds.

The algorithm, however, has some serious disadvantages. It requires two communication messages for each subproblem expansion. The first is required to send the subproblem from the host to the node for expansion. The second is needed to carry the newly created subproblems from the node to the host. Communication with the host becomes a bottleneck that reduces the performance of the algorithm. Also because the processors are unsynchronized, communication delays can lead the algorithm into examining subproblems that need not be examined. The algorithm also requires a large memory on the host processor to maintain the list of active subproblems.

4.2 The Distributed List Algorithm

In the CL algorithm, the use of resources of the hypercube multiprocessor is not balanced nor efficient. The global data is stored in the host memory leaving the memory of the processing nodes unutilized. Also, host to node communication is a bottleneck while the communication bandwidth available among the processing nodes is not utilized. The DL algorithm attempts to put the resources of the hypercube to better use by distributing the global data across the processing nodes.

The DL algorithm consists of $N+1$ processes. Each process maintains its own set subset of the global data. That is, each process maintains its own list of active subproblems and its own incumbent. The first of the $N+1$ processes is referred to as the supervisor process. It initiates the computation by generating N subproblems and assigning one subproblem to each of the remaining N processes. In other words, each process is assigned a subtree of the BB tree that is rooted at the subproblem it received. Each process then expands that subproblem and all subproblems in the subtree rooted at it. Each process expands subproblems from it own local list, performs the lower bound test using its local incumbent, performs feasibility and termination tests, and inserts the results back on its local list. In our implementation, the host runs the supervisor process while each of the N processing nodes run one of the other processes.

Two major problems result from the distribution of the global data across the processes and affect the performance of the DL algorithm. The first is that the N subproblems selected by the N processes are not necessarily the N subproblems with the best lower bounds in the global sense. Although this does not affect the correctness of the results of the algorithm, it does increase the number of subproblems examined by the algorithm. This reduces the efficiency of the parallel BB algorithm and, hence, reduces the speedup of the DL algorithm.

The second problem is the load imbalance that can occur. Due to feasibility, bounding and termination tests, the number of subproblems expanded varies from one process to the other. Eventually some processes run out subproblems and become idle and hence, reduce the speedup of the algorithm.

Therefore, we employ a mechanism by which the load can be balanced and the subproblems be distributed across the processors to approximate the selection of the best N subproblems each iteration. A process expands subproblems from its list until the list becomes empty. At this point, the process becomes idle and requests subproblems from one of the other processes in the system. The process that receives the request examines its own list of active subproblems and either sends a portion of it to the requesting process or denies the request if its own list is too small to divide. In our implementation on the hypercube multiprocessor, a processor requests subproblems from one of its neighbors in the hypercube topology. A processor send one half of its subproblems to an idle processor requesting subproblems.

The distribution of the global data also results in multiple copies of the incumbent. Processes can find feasible solutions independently and update their own incumbents. In the DL algorithm, once an incumbent is updated, its new value is broadcasted to all other processes.

There is also the need to detect the termination of the algorithm in a distributed manner. The detection should require minimal overhead. We employ a heuristic approach that is motivated by the phases of execution of a BB algorithm shown in Fig. 3. Termination need only be checked during the wind-down phase of execution. During that phase, the number of subproblems in the system is very small and not all the processors need to participate in the algorithm. Therefore, an idle process that can not obtain subproblems from its all neighbors terminates sending a message to the supervisor process. When all processes terminate the supervisor process terminates the algorithm.

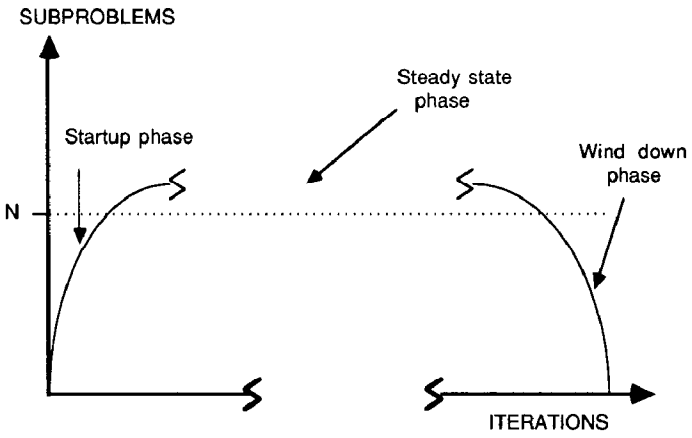

Figure 3: Phases of execution of a BB algorithm.

5 Results

The two algorithms were implemented in the C programming language on an NCUBE/six [NCUB85]. Figure 4 shows the speedup of the two algorithms for various cube sizes. The speedup of the DL algorithm is shown with and without load balancing.

In the CL algorithm, the speedup is reasonable for up to 16 processors. Little is gained by increasing the number of processors beyond that. This can be attributed to two factors. The first is the host to processor communication overhead that increases as the cube size increases. In the 0–1 ILP problem, the ratio of that overhead to the computation time for a subproblem is somewhat high. However, in other problems, like the general ILP problem for example, this ratio can be small and good speedup can be maintained for larger cubes sizes. The second factor relates to the fact that as the cube size increases, some processor become farther away from the host, resulting in an increased delay in communicating messages between the host and these processors. Consequently, these processors receive less subproblems from the host to expand than the processors that are closer to the host. This creates a load imbalance that reduces the speedup.

The speedup of the DL algorithms with no load balancing is constant and extremely low for all cube sizes. This is basically due to the distribution of the list of active subproblems across the processors without any consideration for load factors. For example, in some cases, over 80% of the processors become idle after expanding only a few subproblems while

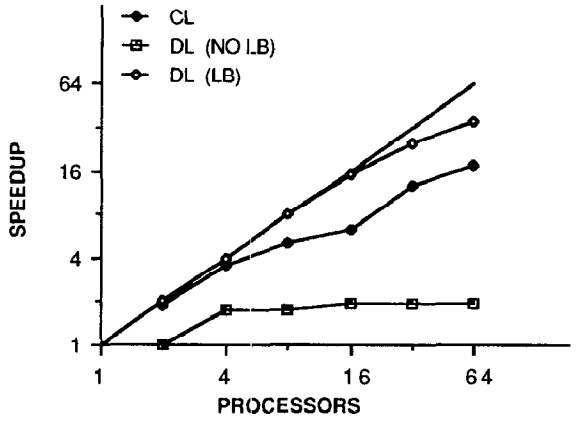

Figure 4: Speedup of the parallel algorithms.

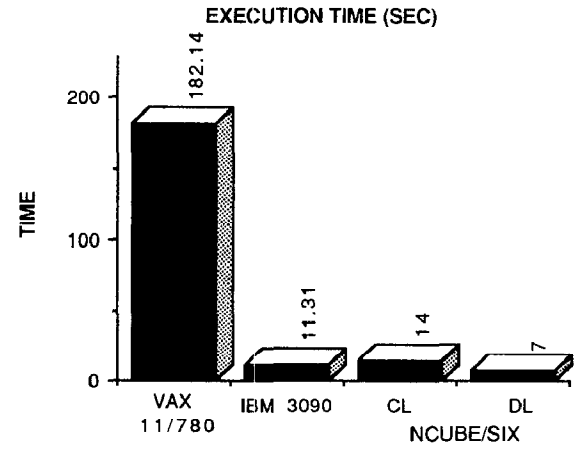

Figure 5: Execution time for various systems.

the remaining 20% end up evaluating the rest of the subproblems in the tree. Therefore, load balancing is necessary. The performance of the DL algorithms with the load balancing shows that a distributed list approach has better performance then the CL algorithm. This is expected since there is no bottleneck in communication; the communication bandwidth of the hypercube is utilized more efficiently.

The performance of the two algorithms at 64 processors is compared to the performance of the serial algorithm on the VAX 11/780 and the IBM 3090 (single processor). The results are shown in Fig. 5.

6 Conclusions and Future Research

Branch and bound algorithms are a generalization of many search algorithms used in Artificial Intelligence and Operations Research. Although BB algorithms are considerably more efficient than unguided search, they are still computationally intensive. Large scale multiprocessors with distributed memory offer the potential for effectively speeding up many computationally intensive applications. However, it is not clear how this class of multiprocessors can be effectively used for BB algorithms.

In this paper, we have considered the implementation of parallel BB algorithms on distributed memory multiprocessors such as hypercubes. Two parallel algorithms for the 0–1 ILP problem were considered. The results indicate that good performance can be achieved using a centralized list of subproblems only for a small number of processors. To achieve better performance, the list must be distributed and a load balancing mechanism must be employed. We have implemented a simple load balancing mechanism that works well for our test problems. However, other preliminary results indicate that the load balancing scheme employed by the DL algorithm may not be effective in some cases. We are currently working towards improving the load balancing scheme [Abde88] for these cases.

References

[Abde88] T. S. Abdelrahman, *Parallel Best First Branch and Bound Algorithms on Distributed Memory Multiprocessors*, Ph.D. Dissertation (in preparation), University of Michigan, 1988.

[AnCh86] S. Anderson and M. C. Chen, "Parallel branch-and-bound algorithms on the hypercube," *Proc. Second Conf. on Hypercube Multiprocessors*, pp. 309–317, 1986.

[FiMa85] R. Finkel and U. Manbar, "DIB – A distributed implementation of backtracking," *Proc. Conf. Distributed Computing*, pp. 446–452, 1985.

[GeMa72] A. M. Geoffrion and R. E. Marsten, "Integer programming algorithms: A framework and

state-of-the-art survey," *Management Sci.*, vol. 18, pp. 465–491, May 1972.

[Ibar76a] T. Ibaraki, "Computational efficiency of approximate branch and bound algorithms," *Math Oper. Res.*, vol. 1, no. 3, pp. 287–298, 1976.

[Ibar76b] T. Ibaraki, "Theoretical comparisons for search strategies in branch and bound algorithms," *Int. J. Comput. Inform. Sci.*, vol. 5, no. 4, pp. 315–344, 1976.

[Ibar77] T. Ibaraki, "The power of dominance relations in branch and bound algorithms," *J. ACM*, vol. 24, no. 2, pp. 264–279, 1977.

[Ibar78] T. Ibaraki, "Depth-m search in branch and bound algorithms," *Int. J. Comput. Inform. Sci.*, vol. 7, no. 4, pp. 315–343, 1978.

[InKo77] G. Ingargiola and J. Korsh, "A general algorithm for one dimensional knapsack problems," *Oper. Res.* vol. 25, no. 5, pp. 752–759, 1977.

[KuKa83] V. Kumar and L. Kanal, "A general branch and bound formulation for understanding and synthesizing AND/OR tree search procedures," *Artificial Intelligence*, vol. 21, pp. 179–198, 1983.

[LaSa83] T. H. Lai and S. Sahni, "Anomalies in parallel branch and bound algorithms," *Proc. Int'l Conf. on Parallel Processing*, pp. 183–190, 1983.

[LaWo66] E. L. Lawler and D. W. Wood, "Branch-and-bound methods: A survey," *Oper. Res.*, vol. 14, pp. 699–719, 1966.

[LMSK63] J.D.C. Little, K. G. Murty, D. W. Sweeney and C. Karel, "An algorithm for the traveling salesman problem," *Oper. Res.*, vol. 11, no. 6, pp. 972–989, 1963.

[Moha83] J. Mohan, "Experience with two parallel programs solving the traveling salesman problem," *Proc. Int'l Conf. on Parallel Processing*, pp. 191–193, 1983.

[NCUB85] NCUBE Corp., *NCUBE Handbook*, version 0.6, Beaverton, Ore., Dec. 1985.

[QuDe85] M. J. Quinn and N. Deo, "An upper bound for the speedup of parallel branch and bound algorithms," *Proc. of the 3rd Conf. on Found. of Software Technology and Theoretical Computer Science*, Bangalore, India, pp. 488–504, 1985.

[QuDe86] M. J. Quinn and N. Deo, "An upper bound for the speedup of best first branch and bound algorithms," *BIT*, vol. 26, no. 1, pp. 35–43, 1986.

[Quin86] M. J. Quinn, "Implementing best first branch and bound algorithms on hypercube multicomputers," Technical report PCL–86–02, Department of Computer Science, University of New Hampshire, Durham, New Hampshire 03824.

[WaLY85] B. W. Wah, G. Li and C. F. Yu, "Multiprocessing of combinatorial search problems," *Computer*, vol. 18, no. 6, pp. 93–108, 1985.

[WuCo80] N. Wu and R. Coppins, *Linear programming and extensions*, McGraw Hill, New York, pp. 420–426, 1980.