N88-17229

PERFORMANCE ANALYSIS OF PARALLEL BRANCH AND BOUND SEARCH WITH THE HYPERCUBE ARCHITECTURE

Captain Richard T. Mraz, USAF
1MSCS/DOXC
Johnson Space Center, Texas 77058

ABSTRACT

With the availability of commercial parallel computers, researchers are examining new classes of problems for benefits from parallel processing. This paper presents results of an investigation of the class of search intensive problems. The specific problem discussed in this paper is the 'Least-Cost' Branch and Bound search method of deadline job scheduling. The object-oriented design methodology was used to map the problem into a parallel solution. While the initial design was good for a prototype, the best performance resulted from fine-tuning the algorithm for a specific computer. The experiments analyze the computation time, the speed up over a VAX 11/785, and the load balance of the problem when using a loosely coupled multiprocessor system based on the hypercube architecture.

INTRODUCTION

Within the past decade, parallel computer architectures have been a subject of significant research efforts. Integrated circuit technology, high speed communications, along with hardware and software designs have made parallel computers much easier to build and much more reliable (6,10,14,15). Parallel processing has also proven to be an effective solution to certain classes of problems. Probably the most notable class is array or vector problems that run order-of-magnitudes faster on parallel architectures such as the Cray. Because of the recent proliferation of parallel computers, researchers are investigating other classes of problems for potential benefits from parallel architectures. Search intensive problems are one such class. Figure 1 illustrates, research in the area of parallel computers has been highly successful in producing several general purpose hardware designs. Clearly, this list indicates the availability of parallel processing system hardware; however, the application and software support systems are not as prevalent. Stankovic points out that "much of the distributed system software research is experimental work" (17: 17). He further emphasizes that "work needs to be done in the evaluation of these systems in terms of the problem domains they are suited for and their performance" (17: 17).

Yet, another area of interest in parallel processing is the mapping of a problem to a parallel solution. Probably, the largest problem researchers face today in parallel computer systems is the inability of humans to decipher the inherent parallelism of problems that are traditionally solved using sequential algorithms. Patton identified a possible cause of this human shortcoming when he said, "While the world around us works in parallel, our perception of it has been filtered through 300 years of sequential mathematics, 50 years of the theory of algorithms, and 28 years of Fortran programming" (11: 34). Basically, humans have not trained their thought processes to accommodate the concepts of solving problems in parallel. Because of this, without new parallel computing algorithms, parallel software development tools, and performance measuring techniques, parallel computing may never be fully exploited.

Company	Product
Alliant Computer Systems Corporati on	FX/Series
Bolt, Beranek, and Newman	Butterfly
Control Data Corporation	Cyber 205 Series 600
Cray Research Inc.	Cray-2 and X-MP
Digital Equipment Corporation	VAX 11/782 and 784
ELXSI (a subsidiary of Trilogy Inc .)	System 6400
Encore Computer Corporation	Multimax
ETA Systems Inc. (a spin-off of Control Data Co rporation)	GF-10
Floating Point Systems Inc.	T Series
Goodyear Aerospace Corporation	MMP
IBM Corporation	RP3
Intel Scientific Computers	iPSC
Schlumberger Ltd.	FAIM-1
Sequent Computer Systems Inc.	Balance 21000
Thinking Machines Corporation	Connection Machine

Figure 1: U.S. Companies Offering or Building Parallel Processors (6:753)

Problem

Because of the proliferation of parallel computers and because a large class of problems that may benefit from parallel processing are search intensive, this research investigated the actual performance of a class of search problems on the Intel iPSC Hypercube computer.

Two examples of the need for this research into parallel search algorithms and performance evaluations are elements of the Strategic Defense Initiative (SDI) and the Pilot's Associate (PA). The SDI Organization is investigating defensive weapon systems and battle management systems for a strategic defense. While researchers for the PA program are investigating flight domain systems that provide expert advice in critical mission functions, such as aircraft systems monitoring, situation assessment, mission planning, and tactics advising (5,12:102,16). The general approach to solve some of the battle management and PA problems uses traditional operations research (OR) and artificial intelligence

(AI) programming techniques. These techniques are based on a systematic search of the solution space of the problem. Hence, this research focuses on parallel search methods. And without loosing generality, the specific technique is parallel branch and bound. For example, the SDI battle management system must resolve the resource allocation of sensor and tracking satellites to defensive weapon systems (16: 4-5). Answers to such a problem involves a complex solution space with exponential computation time to find the optimal solution. Researchers plan to reduce the run time complexity using parallel computers. The ultimate goal is to find the proper combination of parallel computer architecture and parallel algorithm such that results can be calculated in "real-time", where "real-time" is that time interval in which an answer must be delivered(10:8).

While a general search definition is useful during the parallel design phases of research, a specific problem must be solved for an actual performance evaluation. To this end, the specific class of 'Least-Cost' Branch and Bound search is used, where the basis of the hypercube performance evaluation is the Deadline Job Scheduling (DJS) problem. In a DJS problem, a set of jobs or tasks are defined by a 3-tuple (p_i,d_i,t_i), where

p_i = Penalty for not scheduling job i
d_i = Deadline by which job i must be completed
t_i = Time to run the job i

The goal is to find the largest subset of jobs that can run by their deadline while minimizing the total penalty incurred. This search uses both a ranking function to identify potentially good solution paths and two bound functions to eliminate needless searching in parts of the solution space. The DJS problem is characterized by exponential time complexity to find the optimal subset of jobs (worse case).

The goals of this research can now be summarized as follows,

1- Explore a design methodology to map a problem into a parallel computer.

Because of the difficulties of mapping a problem to a parallel computer, a formal design approach is needed to help the programmer identify the parallel activity within a problem. Since the development and proof of a new design methodology is beyond the scope of this research, only traditional design approaches will be examined.

2- Measure the performance of parallel branch and bound search on a parallel computer.

Since some researchers with search intensive problems, such as the SDIO and Pilot's Associate, have requirements for 'real-time' processing, experiments must be run to examine the possibilities for speed up. The results of a parallel branch and bound test can be used as a benchmark for further research as well.

3- Evaluate the hypercube as a suitable architecture for search algorithms.

In conjunction with the development of a good parallel algorithm, the speed up of a problem is also a function of the parallel computer architecture. Therefore, as Stankovic pointed out, the parallel architecture must be evaluated to identify their suitable problem domains.

Parallel Processing Issues

Two fundamental issues of parallel processing form a basic set of constraints for parallel problem solving. Simply stated, the first concept of maximum parallelism places a restriction on a parallel solution. This constraint may take several forms. First, the problem may inherently have limitations and dependencies that cannot be overcome. Second, a poor algorithm may inhibit parallel activity. Finally, parallel computer architectures have been targeted to solve specific classes of problems.

The second parallel processing issue deals with the mapping of a problem into a parallel solution. For humans, thinking in parallel does not come naturally. Therefore, a design methodology is needed to describe a problem such that parallel activity can be identified.

Overview of the Paper

In the introduction, a look at the need for this research, the definition of the problem, and the description of two parallel processing issues identified fundamental concepts used throughout this research. In the next section, a description of the hypercube computer presents the parallel environment for this research. Then, the definition of search and the parallel branch and bound design is reviewed. Following the design, the experimental results and conclusions of this research complete the paper.

HYPERCUBE ARCHITECTURE

The parallel environment for this research is the Intel iPSC Hypercube computer. Initial research on the hypercube, known as the Cosmic Cube, was conducted by Professor Charles L. Seitz at the California Institute of Technology (8,14). The basis of the hypercube computer can be described by the process model of computation (14). Simply stated, the process model describes the interaction of processes using message passing instead of shared variables(14:22). Using such a model, "a programmer can formulate problems in terms of processes and 'virtual' communication channels between processes" (14:23). The Intel iPSC hypercube used in this research adheres to the process model of computation in two ways. First, programmers define and encapsulate processes on any iPSC node. In fact, several processes can be placed on each iPSC node. Second, the iPSC operating system provides a set of message passing primitives for interprocess communication. The processor interconnection strategy that provides good message passing properties to support this model of computation is called the binary-n-cube or hypercube (see Figure 2) (14,15,18). As described by Wu, the binary n-cube is a network of 2^n processors where each node has n neighbors (18:239). The number n also describes the dimension of the cube. For example, a 3-dimension cube has 2^3 nodes and each node has 3 neighbors. Node identification consists of a binary number of length n (see Figure 2).

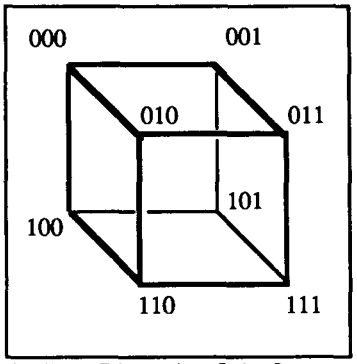

Figure 2: Three-Dimension Cube Structure, with vertices labeled from 0 to 7 in binary (15:66).

In addition to the message passing architecture, a programmer can configure the hypercube into several logical structures, such as ring, tree, grids, torus, and bus using specific message passing schemes (8,14). Using these structures, efficient nearest neighbor communications is maintained and the structure of the parallel solution can be designed to match the structure of the problem.

FUNDAMENTALS OF SEARCH

Search is a basic Operations Research (OR) and Artificial Intelligence (AI) programming technique. Such a strategy is used when problems cannot be solved using direct methods (i.e. formulas, algorithms, etc.) (13:55). Several specific search strategies have been developed (2,7,13). Each strategy varies the way the solution space of the problem is examined for answers. Sometimes the entire solution space is blindly searched for an answer. While other search techniques use heuristics or rules to guide through the solution space. The solution space for a search is typically represented using a tree organization (7:325). Horowitz and Sahni describe the search tree as follows (7:325-329). The root of the tree represents the *initial state* of the problem (see Figure 3). Each nonterminal node in the tree represents a *problem state* in the search. The *state space* of a search is defined as the collection of all paths from the root node to any node in the tree.

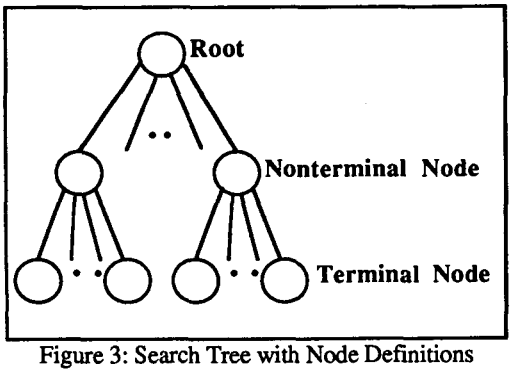

Figure 3: Search Tree with Node Definitions

Even though trees are used to represent the solution space of a search, the tree is usually not stored explicitly in the computer. Because search problems have the additional overhead of combinatorial explosion due to the branching factor or the depth of the tree, only portions of the tree needed to solve the problem are kept in storage. For this research, branch and bound, the general form of a state space search, is used. By manipulating two functions, a ranking function and a bound function, branch and bound can be used to model 'blind' as well as intelligent search. While 'blind' search techniques, such as Depth-First and Breadth-First search, do not use knowledge of the problem domain to control the search process, other search methods, called intelligent search, try to narrow the search space, shorten the search time, and reduce the storage needed by applying knowledge of the problem domain to control the search. The following actions are used to meet the three goals of 'intelligent' search (2:59),

1- Decide which node to expand next.
2- Select the most promising successors when expanding a node.
3- Eliminating or pruning the search tree.

To represent a node in the branch and bound search space, a solution vector $(x_1,x_2,\ldots,x_n)$, is used (7:323). Each x_i is constrained by explicit and implicit constraints. The explicit constraints define the range of values that each x_i can be assigned. For example, the solution vector for a 4-Task Deadline Job Scheduling problem is (x_1,x_2,x_3,x_4). The explicit constraints for this problem are simply $x_i \in \{0,1\}$, where 1 denotes that task i is included in the schedule and 0 denotes that job i is not included in the schedule. For instance, a valid solution vector for the 4-Task problem would be (1,1,1,•). This vector represents the search state where jobs 1, 2, and 3 have been scheduled and job 4 has not been scheduled. To help the reader in understanding the DJS constraints,, the following job set will be used in examples throughout this section (16:384),

Job	p_i	d_i	t_i
1	5	1	1
2	10	3	2
3	6	2	1
4	3	1	1

Using the definition of explicit constraints, the solution space of the example job set is depicted in Figure 4. The grey node identifies the example solution vector (1,1,1,•). The second set of constraints, implicit constraints, define relationships among the various x_i's. The nodes in the solution space that meet both the explicit and the implicit constraints define answer nodes. The first implicit constraint for the DJS problem is called the Deadline/Total Time Bound. This constraints requires a job to be scheduled such that the total run time for all jobs included in the schedule does not exceed the maximum deadline. Referring to the example 4-Job problem above, the solution vector (1,1,•,•) passes the Deadline/Total Time implicit constraint because the maximum deadline of jobs 1 and 2 is 3 and the total run time of jobs 1 and 2 equals 3. However, the solution vector (1,1,1,•) does not pass the Deadline/Total Time Bound because the maximum deadline of jobs 1, 2, and 3 equals 3 and the total run time of those same jobs equals 4.

The second implicit constraint, Cost/Upper Bound, for the DJS problem is based on the cost of the node and a global upper bound. The cost function is calculated in two steps (16:386). First, find **m** where,

$$\mathbf{m} = \max\{i \mid i \in S_X\}$$

S_X = the subset of jobs examined a node X

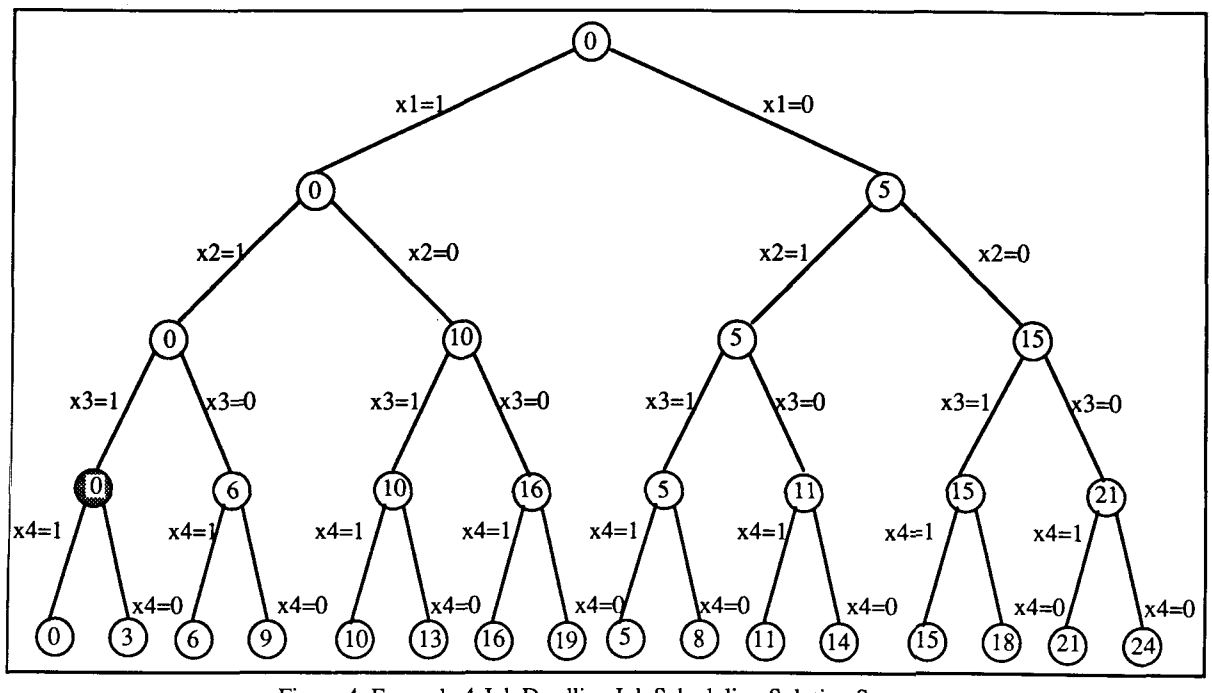

Figure 4: Example 4-Job Deadline Job Scheduling Solution Space

Next, compute the cost of node X using the following equation,

$$c'(X) = \sum_{\substack{i \le m \\ i \notin J}} p_i$$

where J = the set of jobs included in the schedule at node X.

The cost of a node translates to the total penalty incurred of all jobs that have not been scheduled so far. The cost of each node of the example job set is shown inside the circles of Figure 4. For example, the cost of solution vector (1,0,•,•) equal 10 because,

$S_X = \{1,2\}$

$m = \max\{i \mid i \in S_X\} = 2$

$J = \{1\}$

$$c'(\ (1,0,\bullet,\bullet)\ ) = \sum_{\substack{i \le 2 \\ i \in \{2,3,4\}}} p_i = 10$$

The second part of the Cost/Upper Bound constraint involves calculating the upper bound of node X using the following function,

$$U(x) = \sum_{i \notin J} p_i$$

The value of the upper bound identifies the maximum cost solution node inthe subtree rooted at node X. For example, vector(1,0,•,•) of the tree in Figure 4 has an upper bound of 14 since the cost of solution node (0,1,0,0) equal 14 and solution node (0,1,0,0) is the highest cost node in the subtree. During the seach, the lowest upper bound is maintained as a global bound. The gobal upper bound is defined by the following function,

global upper bound = min{U(x),current upper bound}

A child of the current node being expanded is added to the list of 'live nodes' (nodes that will be expanded later) if the cost of the child is less than the global upper bound. (Note: the list of 'live nodes' is maintained in Least-Cost order; hence, the name Least-Cost Branch and Bound).

PARALLEL DESIGN

One goal of this research was the investigation of a parallel design methodology. After reviewing several common design strategies, the object-oriented design methodology was selected for the research (3,4). The basic concept of an object design is the decomposition of the problem into objects, operations, and communications among the objects. Because the hypercube architecture is defined by the process model of computation, where 'processes' communicate using 'messages', parallel solutions defined by an object design map naturally into the hypercube. The programmer can describe the problem as fine-grained objects using the object model. These objects can be mapped directly to hypercube processes, or for efficiency and reduced communications, a collection of objects can be implemented as a hypercube process. Figure 5 shows the configuration of the parallel branch and bound search objects and the communications dependencies among those objects. The first process is called the Control Process. It is defined by the objects in the top processor box (see Figure 5). The meta-controller, terminate check, and bound check serve as global control throughout the parallel search. The Control Process resides in Node 0 of the hypercube. The remaining nodes contain the Worker Process. The task of each Worker Process is to find the best answer to a subproblem. A subproblem for a branch and bound search is equivalent to searching a subtree of the solution space for the best answer in that subtree.

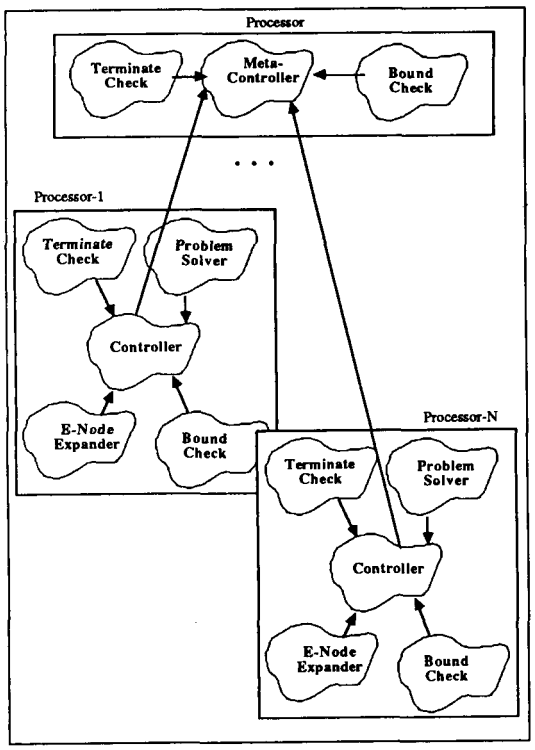

Figure 5: Object Visibility Diagram

During the search, the Control Process monitors the progress of the search by creating an initial set of subproblems to solve, sending those subproblems to Worker Processes, and terminating the search. Upon receiving a problem, the Worker Process finds the best answer (in that subtree). Once the entire subtree has been examined, the Worker Process posts a 'work request' to the Control Process and waits for additional work. The entire problem is finished when the Control Process does not have any problems to solve and all Worker Processes have posted 'work requests'. This translates to a machine state where no more work is available and all workers need a problem to solve.

EXPERIMENTAL RESULTS

Three measures, Computation Time, Speed Up, and Load Balance, were used to categorize the performance of the parallel search problem on the iPSC Computer. First, Computation Time measures the run time of a problem. Because the parallel computational environment involves additional processes, one representation of the total computation time of an algorithm is defined by the following formula (1:95),

$$T_N = T_S + T_C + T_W \qquad (1)$$

where

T_N = Computation Time for N processors
T_S = Start Up Time
T_C = Processor Computation Time
T_W = Wind Down Time

Start Up Time, T_S, measures the time to initialize the parallel processor before any parallel computation begin. Start up may include such things as initial parsing of the job, initial message transfers, or down load time of the programs to the parallel machine itself. The second term, processor computation time, measures the time the computer spends actually solving the problem. This term is common in sequential processor run time analysis. The final term in equation 1 is wind down time. This time accounts for the gathering of results from the various processors in the computer and analyzing or tallying those final results.

The second measure for the performance evaluation, Speed Up, compares the time to compute a solution using one processor and the time to compute a solution using N processors. It is defined as follows (14:28),

$$S = T_1 \div T_N \qquad (2)$$

where

T_1 = Time to computer a result with one processor
T_N = Time to computer a result with N processors (Eqn 1)

The speed up of a problem run in parallel is intuitively easy to understand. If a problem can be parsed into N sub-problems, with each subproblem taking 1/N of the total computation, then the maximum speed up of N is achieved. The perfect speed up, N, is highly unlikely because of the overhead of start up and the wind down time. Communications among processing elements also induce limitations on this measure.

Finally, the third measure, load balance, may be helpful in identifying computation bottlenecks. Because of the nature of the design and the branch and bound problem, the 'load' is defined to be the number of nodes expanded by a Worker Process. When plotted against the average load performed across all Worker Processes, balanced and unbalanced work loads can be identified. Single-Instruction-Multiple-Data (SIMD) problems that partition data to promote parallel activity tend to have regular communication and computation cycles. These classes of problems show the best performance under balanced work loads (9). Since parallel search is a Multiple-Instruction Multiple-Data (MIMD) problem, the communication and computation cycles cannot be guaranteed to be regular. Hence, the load balance measure must be evaluated along with the other performance measures before drawing conclusions.

Baseline Performance

The baseline of performance for this research is a Digital Equipment Corporation VAX 11/785 running the 4.2 BSD (Berkeley Software Distribution) UNIX operating system. The configuration of the machine used for this research has 8 Megabytes of main memory and 1800 Megabytes of disk storage. The sequential versions of the Deadline Job Scheduling problems were programmed in C Language, and the time information was obtained using the UNIX "times" function. Of the four parameters measured by the "times" function, this research focused on user_time. The user_time of a process is that time devoted to computation. The overhead associated with system calls, page swaps, etc. was not used for two reasons, (1) this research is actually interested in compute time of the algorithm and not operating system overhead; and (2) the VAX is under various system loads during the course of the experiments which would influence system time and the overall timing data.

Parallel Performance

First, the DJS problem was tested on the Intel iPSC simulator running on the VAX. While the simulator creates a good environment to learn how to program the iPSC, it does not show true parallel activity. Hence, it should not be used to fine-tune a problem. After porting the code from the

simulator to the actual iPSC, the original design was modified to achieve the best computation times. It should be noted that the object design worked well for an initial implementation, but the best performance results were attributed to fine tuning on the actual hardware. The only part of the parallel DJS used for fine-tuning was the iPSC Control Process. This process has the responsibility to create the initial set of problems to solve. At some point, it becomes beneficial to stop creating problems and to start handing them out to worker nodes. It should be noted the results of these experiments have fine-tuned to large problem sizes. For the parallel experiments, job sets from 4-Jobs to 25-Jobs were tested on six cube sizes, d-1, d-2, d-3, d-4, and d-5, where d = dimension). The d-0 cube could not create a data structure large enough to solve large DJS problems. Timing results for all runs was calculated using the iPSC Clock function on each node of the hypercube. Since the resolution of the iPSC Node Clock function is 1/60th of a second, some of the computation times were unmeasurable. The data in this section has been plotted for comparison and to show trends.

Before analyzing the results of the job scheduling experiments, a description of the test data is necessary. Since the deadline job scheduling solution uses least-cost branch and bound, them time to schedule a set of **n+1** jobs may take less time than scheduling **n** jobs. Therefore, two pseudo-equivalent classes of problems were devised such that the larger the job set created a more difficult problem to solve. Two reasons for creating pseudo-equivalent classes are, (1) the proof of equivalent classes of jobs is beyond the scope of this research; and (2) job set with these characteristics make the analysis a bit easier.

As described in a previous section, each job is defined by a 3-tuple (p_i,d_i,t_i), where p_i is the penalty incurred if the job is not scheduled, d_i is the deadline when the job must be finished running, and t_i is the time to run job i. With this information, the first set of problems (see below) guarantees that all jobs can be scheduled. The VAX solves this problem in O(n) time.

$$p_i = 1, \forall\ i$$

$$\sum_{i=1}^{n} t_i \leq \min(d_i)$$

The second pseudo-equivalent class is solved in exponential time by the VAX and it is described with the following values for the job 3-tuples,

$$t_i = i$$

$$p_i = 2^{*}t_i$$

$$d_i = \left\lfloor \frac{\sum_{i=1}^{n} t_i}{2} \right\rfloor$$

First, an analysis of the O(n) job set. Parallel processing appears to show no reductions in the time order complexity of O(n) problems (see Figure 6). The best performance was attributed to the iPSC d-1 and the best speed up was approximately 0.33 over VAX. Since this search problem degenerates to an examination of the left-most branch of the search tree, the problem does not map well to a parallel processor. The Load Balance analysis shows this result (see Figure 7). Basically, this problem cannot run in parallel. For small problem sizes, (scheduling 15 jobs or less) only one processor solves the problem while for large problem sizes, two iPSC worker nodes are used. This problem re-enforces the concept of maximum parallel activity because of limitations inherent to the problem.

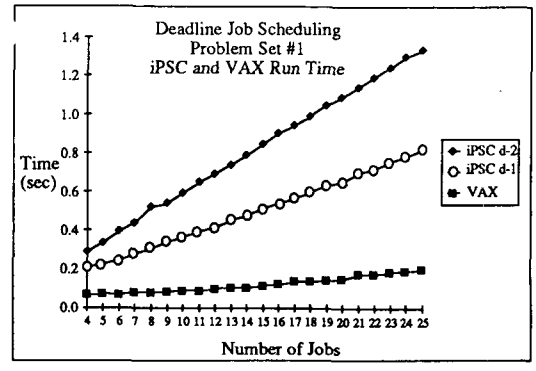

Figure 6: Deadline Job Scheduling- Problem Set #1
Computation Time

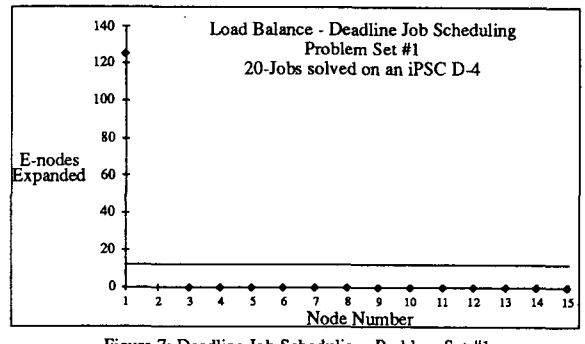

Figure 7: Deadline Job Scheduling- Problem Set #1
Load Balance of scheudling 20-Jobs on an iPSC D-4

Next, the second job set must be examined. The run time analysis shows that significant reductions in the time order complexity can be achieved. In Figure 8, the semi-log plot of the VAX computation time shows the inherent exponential nature of the problem. The iPSC d-4 and d-5 curves demonstrate the power of the parallel computation with problem sizes of 11 or greater. It should be noted the best speed up achieved was 58 times over VAX with a d-5 hypercube solving a 15-Job problem. The d-4 reached a speed up of 43 times over VAX (see Figure 8). A dimension-3 cube attained a speed up 33 times over VAX. Finally, d-2 and d-1 hypercubes solved the problem approximately 3 times faster than VAX. These results can be explained while examining the global upper bound during the parallel computation. Figure 9 shows the load balance of scheduling 15 jobs on a d-4 cube. Even though the load is unbalanced, the iPSC solved this problem 43 times faster than the VAX. Solving this same problem on a d-5 cube, all Worker Processes have node expansion counts (loads) equal to zero. This anomaly is attributed to the global upper bound. In a d-4 cube, the Control Process generates 64 initial problems. As the workers solve these problems

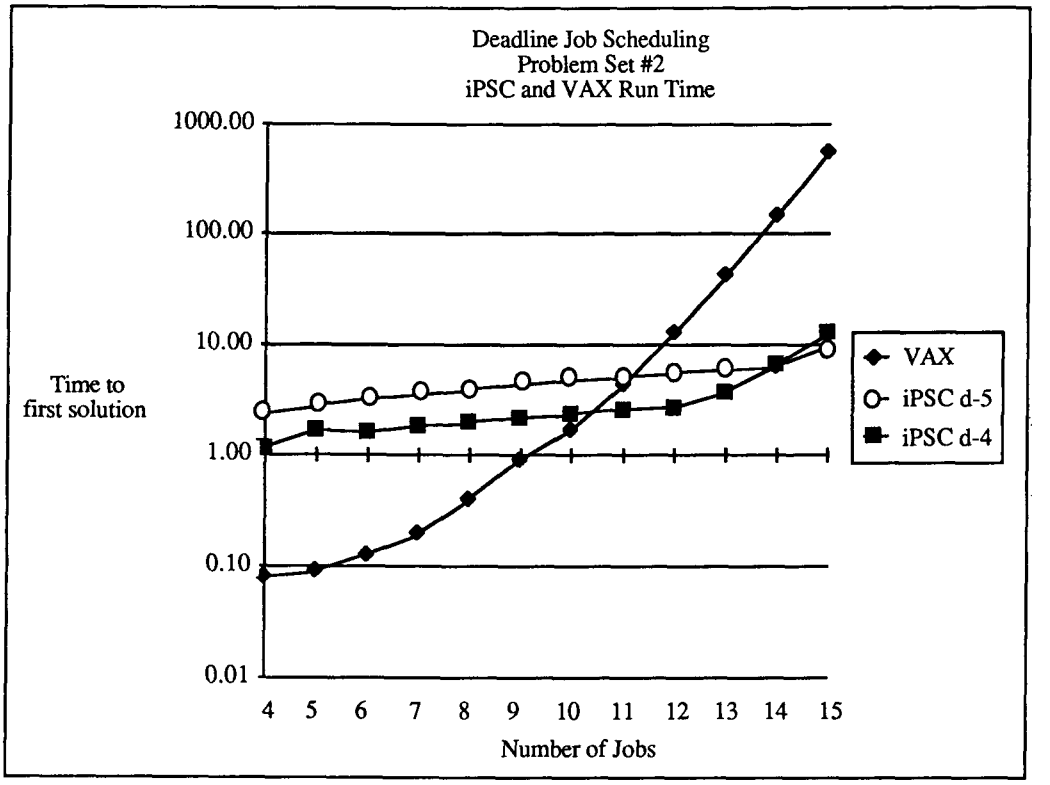

Figure 8: Deadline Job Scheduling- Problem Set #2
Computation Time

concurrently, the global upper bound converges quickly to the best upper bound of the entire search space. Once the upper bound converges, the workers no longer search the subtrees. They only prune the remaining search space. In the case of the d-5 hypercube, the Control Process generates 128 initial problems to solve. At this point the upper bound has all ready converged, and the search quickly ends with the workers just pruning the search space and never actually searching the subtrees.

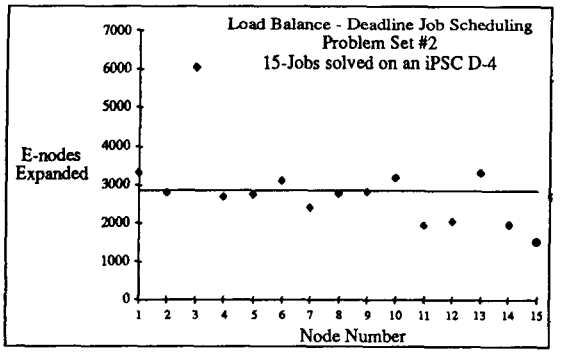

Figure 9: Deadline Job Scheduling- Problem Set #2
Load Balance of scheduling 15-Jobs on an iPSC D-4

CONCLUSIONS

The main objective of this research was the performance evaluation of search problems on the hypercube architecture. First, conclusions from the other research goals. The results of the first goal identified the object-oriented design methodology as a good design approach to map a problem into a parallel solution. The object model worked well for this research. The results of the object design resulted in a fine grained mapping of the problem space, and the implementation of the design focused on collecting several objects into coarse grained iPSC processes. During the design, details of the branch and bound problem were not overlooked and during the implementation, inefficiencies of communications were reduced. Even though the initial design needed fine tuning to achieve the best performance, the implementation of the initial design created a good prototype. As a recommendation for future research, the object design methodology should be extended to other parallel processors such as shared memory machines or other hypercube architectures. As noted with this research, an object design worked well for the hypercube because of the similarity of object design and the process model of computation. Additional tests of object-oriented design will test the flexibility and suitability of the design methodology as a general approach to map a problem into a parallel architecture.

The second goal of this research was the performance evaluation of search problems on a parallel processor. As the results show, a sequential problem solving technique, like search, can be mapped to a parallel processor and speed ups over traditional sequential machines can be achieved. In fact, over a narrow range, the parallel solution reduced the time order complexity of the problem. But, the results of the O(n) job set also re-enforced the concept of maximum parallel activity due to limitations within the problem.

C-3

Finally, the third goal of this research was to examine the suitability of the hypercube architecture to solve search problems. Branch and bound is a sequential programming technique with centralized control. The parallel solution presented in this paper was mapped onto an extremely loosely coupled architecture. Even though this research successfully produced speed ups, the nature of the hypercube architecture and the nature of the problem are not similar. Therefore, parallel search should be examined on other, more tightly coupled architectures, such as shared memory machines. Yet another approach to speed up search problems is to design new algorithms instead of mapping sequential techniques to parallel processors.

BIBLIOGRAPHY

1. Akl, Selim G. et al. "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm," IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-4: 192-203 (March 1982).

2. Barr, Avron, and Edward A. Feigenbaum. The Handbook of Artificial Intelligence, Vol 1. Standford, California: HeurisTech Press, 1981.

3. Booch, Grady. "Object-Oriented Development," IEEE Transactions on Software Engineering, SE-12: 211-221.

4. Booch, Grady. Software Engineering with Ada. Menlo Park, California: Benjamin/Cummings, 1983.

5. Bosma, John T. and Richard C. Wheelan. Guide to the Strategic Defense Initiative. Arlington, Virginia: Pasha Publications, 1985.

6. Fenkel, Karen A. "Evaluating Two Massively Parallel Machines," Communications of the ACM: 29 752-758 (August 1986).

7. Horowitz, Ellis, and Sartij Sahni. Fundamentals of Computer Algorithms. Rockville, Maryland: Computer Science Press, 1978.

8. Intel iPSC Concurrent Programming Workshop Notes, Intel Scientific Computers, Beaverton, Oregon. (16-20 June 1986).

9. Lee, Lieutenant Ronald. Performance Comparison and Analysis of State of the Art Machines. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1986.

10. Norman, Captain Douglas O. Reasoning in Real-Time for the Pilot Associate: An Examination of a Model Based Approach to Reasoning for Artificial Intelligence Systems using a Distributed Architecture. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1985.

11. Patton, Peter C. "Multiprocessors: Architectures and Applications," Computer: 29-40 (June 1985).

12. Retelle, LtCol John P. Jr. "The Pilot's Associate-Aerospace Application of Artificial Intelligence," Signal: 10-105 (June 1986).

13. Rich, Elaine. Artificial Intelligence. New York: McGraw-Hill, 1983.

14. Seitz, Charles L. "The Cosmic Cube," Communications of the ACM: 28 22-33 (January 1985).

15. Siegel, Howard Jay, and Robert J. McMillen. "The Multistage Cube: A Versatile Interconnection Network," Computer: 65-76 (December 1981).

16. Seward, Walter D., and Nathaniel J, Davis IV. "Opportunities and Issues for Parallel Processing in SDI Battle Management/C3." Presented at the AIAA Computers in Aerospace V Conference, October 1985.

17. Stankovic, John A. et al. "A Review of Current Research and Critical Issues in Distributed System Software," Distributed Processing Technical Committee Newsletter, 7: 14-47 (March 1, 1985).