01 Jul 1985

A Parallel Branch and Bound Algorithm for Integer Linear Programming Models

Rochelle L. Boehning

Billy E. Gillett
Missouri University of Science and Technology

A PARALLEL BRANCH AND BOUND ALGORITHM FOR INTEGER LINEAR PROGRAMMING MODELS

Rochelle L. Boehning* and Billy E. Gillett

CSc-85-2

Department of Computer Science
University of Missouri-Rolla
Rolla, Missouri 65401 (314) 341-4491

*This report is substantially the Ph.D. dissertation of the first author, completed July 1985.

ABSTRACT

A parallel branch and bound algorithm is developed for use with MIMD computers to study the efficiency of parallel processors on general integer linear programming problems. The Haldi and IBM test problems and a System Design model are used in the implementation of the algorithm. Initially the algorithm solves the Haldi and IBM test problems on a single processor computer which simulates a multiple processor computer. The algorithm is then implemented on the Denelcor HEP multiprocessor using two of the IBM problems to compare the results of the simulation to the results using an MIMD computer. Finally the algorithm is implemented on the HEP using the System Design model to show a case in which the number of pivots decreases as the number of processes are increased from seven to the process limit of sixteen.

In general, it is shown that super linear efficiency can be achieved using multiple processors.

ACKNOWLEDGEMENTS

I wish to thank my advisory committee for their help and guidance throughout my graduate work in computer science. They have been much more than an advisory committee, they have also been my teachers, supervisors, colleagues and friends. A special thanks goes to the Praters for opening their home to me and giving much needed emotional support. My advisor, Bill Gillett, is very special, having kept me on the path of my program and never forgetting to encourage me when problems arose.

In 1966, John DeCicco of Illinois Institute of Technology convinced me that I could do research. His patience and guidance kept alive in me the hope of finally obtaining the Ph.D.

Thanks to Ralph Butler for his work on the HEP which kept me from having to reinvent the wheel.

Finally, I must thank my family for their understanding, encouragement and support. Without the strength of my wife, this endeavor would have been impossible.

TABLE OF CONTENTS

TABLE OF CONTENTS CONTINUED

TABLE OF CONTENTS CONTINUED

LIST OF TABLES

I. INTRODUCTION

A. STATEMENT OF THE PROBLEM.

The solving of integer linear programmimg models using the simplex method with a branch and bound algorithm lends itself naturally to implemention on a multiprocessor computer. The parallel implementation of components of a branch and bound algorithm on a multiprocessor computer gives rise to the possibility of achieving super linear efficiency. This means that n processors working in parallel can solve a given problem in fewer total operations than a single processor.

This paper describes a parallel branch and bound algorithm for solving integer linear programming models. The algorithm uses a combination of parallel branch and bound techniques with and without parallel hyperplane cuts. Problems with some (0,1) variables were also investigated using a combination of the above techniques in combination with explicit enumeration. The algorithm was initially implemented using a single processor to simulate many processors working in parallel. This process was used to investigate the IBM and Haldi test problems [1] to demonstrate that parallel processors could achieve super linear efficiency. These problems were chosen since they were designed to test ILP algorithms, are well known and are considered small but difficult. The algorithm was then

implemented on a multiple instruction stream, multiple data stream computer. The IBM-3 and IBM-4 test problems and a System Design model were chosen for investigation on the parallel processing computer to compare with the simulation results.

B. TECHNIQUES FOR SOLVING THE PROBLEM

The simulation was done using the PL/I language on the IBM 4381 computer. The computer simulated was a Multiple Instruction stream, Multiple Data stream (MIMD) type computer with a common memory as well as individual memories in each processor.

The algorithm was then programmed in the "C" language and implemented on the Denelcor HEP multiprocessor at Argonne National Laboratory. The HEP is a general purpose computer that can handle multiple instruction streams and multiple data streams (MIMD) [APPENDIX A]. The macros used to implement the parallel algorithm were adapted from the FORTRAN macros written by Lusk and Overbeek [2,3] and the "C" macros written by Butler [4].

II. LITERATURE SEARCH

The fields of parallel processing and integer linear programming (ILP) have, until recently, been studied by separate groups. This is evidenced by the fact that most of the literature on parallel processing is in the electrical engineering journals with very few articles concerning parallel processing appearing in the operations research journals. The number of papers published in the Proceedings of the International Conference on Parallel Processing has more than doubled in the past five years. The early parallel computing machines which were built in the seventies were of the Single Instruction stream, Multiple Data stream (SIMD) type. Most of the commercially available multiprocessor computers today are still of this type [5].

The operations research community had been disappointed with the applicability of the SIMD type of super computer to mathematical programming (MP) in general and to linear algorithms in particular [6,7,8]. The hope now is that the Multiple Instruction stream, Multiple Data stream (MIMD) [8] computers will help the field of MP. The main advantage MIMD computers have over SIMD computers in solving problems with systems of linear equations, is that the pivot step can be done in parallel with column operations [7]. Present MP applications are mostly in the area of matrix decomposition algorithms [6,7,9,10,11,12] and partial differential

equations [13], with some in graph theory [14].

Techniques for obtaining parallelism include, dividing a single execution string into several concurrent "threads" [15] and dividing programs into sections that "reflect the logical structure" of the problem concerned [16,17].

Working with a parallel computer and global variables cause the concepts of mutual exclusion to be much more critical than in a sequential computer, and hence the use of locks, semaphores and monitors cannot be left to the operating system [17,18,19]. These topics will be discussed later in the paper.

Although the Branch and Bound algorithm for ILP seems to lend itself very naturally to parallelism, the work with this method has been restricted to the area of (0,1) implicit enumeration problems. Two of these papers are of special interest, Lai [20] and Gehringer et.al.[11].

Lai deals with anomalies in the knapsack problem and the travelling salesman problem. Anomalies are defined in the following manner: If n processors take I(n) iterations to do a particular problem and m processors take I(m) iterations for the same problem and $n \leq m$, the inequalities $I(n)/I(m) \leq m/n$ and $I(n) \geq I(m)$ should hold. If either of these inequalities does not hold it is said to demonstrate anomalous behavior. Lai found that in the knapsack problem that anomalous behavior occurred in ten percent of the tests, and that no anomalous behavior occurred in the travelling salesman test problems. He found, in the

knapsack problem, that the speed-up ratio I(n)/I(2n), (i.e. doubling the number of processors), varied from 0.15 to 14.6. Since the number of processors was doubled, the limits should have been 1.0 (i.e. taking as many iterations on twice as many processors) and 2.0 (i.e. taking half as many iterations using twice as many processors). He also felt that an acceptable ratio would have been 1.6. Although it was not stated, it appears that this was to make up for added switching, communication and blocking time due to the extra processors. The binary state space tree was used, since it was believed that the use of an n-ary state space tree would have taken weeks of computer time to complete the simulation. Iterations were used as measurements but were referred to as time measurements, and since no parallel machine was mentioned, it is assumed that the work was done as a simulation.

Gehringer, Jones and Segall [11] reported on how the Carnegie-Mellon group [10], especially Raskin, used the Cm* [21] to compare the Cm* used as a multiprocessor and the Cm* used as a network. The Cm* consists of several mini-processors linked by intercluster busses. This architecture makes the computer behave more like several separate computers than a single multiprocessor. The problem was a set partitioning integer problem using an enumeration algorithm that performs an n-ary tree search in a large, relatively sparse binary matrix for a min-cost solution. The matrix was two dimensional with a size usually in the

order of hundreds by thousands. Greater-than-linear speedup was obtained with a 10-processor Cm*. The speed-up ratio is the time for one processor to complete a task divided by the time for n processors to do the same task. For n processors to achieve at least linear speed-up, this ratio should be at least n. Greater-than-linear speed-up in this case means that one processor will take more than n times as much time to complete a task as n processors will take doing the same task. In this algorithm's initalization phase, a large number of possible solutions are put in a global stack, from which all the processors choose their work. As the search proceeds, the cost of the best solution found so far by any processor is stored as a global variable. All processors compare their current cost value to it and begin to backtrack in the search when the global cost is lower. The multiprocessor could be "lucky", in that one of its processors might encounter a near-optimal solution at the outset and then none of the processors would have to do very much work. The uni-processor version, which does not encounter the near optimal solution until later, has the disadvantage of having done a more complete search over the earlier possible solutions. The multiprocessor could also be "unlucky" if at the outset the near optimal solution is encountered by the uniprocessor. This would cause both the uniprocessor and the multiprocessor to enter at the same time and therefore before the cost could be determined, the other processors in the multiprocessor version would have

wasted processing time on their initial solutions.

The conclusion concerning the ILP test runs was, that although greater-than-linear speed-up was obtainable, it will not usually be obtained. Greater-than-linear speed-up was obtained in one of the five integer programming runs using two through eight processors. All five runs produced approximately linear speed-up for two processors. For eight processors, the worst speed-up was 5.5 and the best was 9.75 where 8 would be linear.

The literature in computer science is void of papers on the use of parallel processors in the solution of ILP problems using simplex type algorithms. The only papers which use parallel branch and bound techniques in ILP were those for the (0,1) type of problem such as AND/OR-tree searches, state space searches, game-tree searches and finding shortest paths in trees [22,23,24,25]. Most of these types of problems use only addition and subtraction since binary matrices are used. Those using weighting factors include some integer multiplication but none use floating point multiplication or division.

The simplex method [26], which is used on more general types of ILP problems, typically uses a large number of divisions on each pivot operation, and there are usually many pivot operations in a problem, hence the computational difficulty is much greater and much more prone to computational errors such as round-off. Taha [27] suggests

that in the general ILP problems, a combination of methods, such as branch-and-bound, cutting plane, and implicit enumeration, may give better results than any one method used alone.

III. PARALLEL BRANCH AND BOUND ALGORITHM

A. INTRODUCTION

Consider the integer linear programming model:

$$\text{Maximize} \quad z = \sum_{j=1}^{p} c(j)x(j)$$

$$\text{subject to} \quad \sum_{j=1}^{p} a(i,j)x(j) \leq b(i) \qquad i=1,2,\ldots,n$$

$$x(j) \geq 0, \text{ integer}$$

where n=number of constraints, p=number of variables, c(j) are the cost coefficients, a(i,j) and b(i) are constants in the constraints, x(j) are the variables and z is the objective function value.

B. THE ALGORITHM

STEP 1.

Initialize Lower Bound (LB) = -10E10 (a number with a large enough absolute value to approximate negative infinity for the problem), state the number of processors and the initial simplex dimensions and then load the initial simplex.

STEP 2.

Calculate the continuous solution using the Lexicographical Column Dual Simplex Method. [27]

STEP 3.

If the continuous solution is an all-integer solution, the problem is solved, so print the results and stop; otherwise, the final tableau of the continuous solution, referred to as a node, is stored as a layer of a three dimensional matrix (node storage) and the z-value (objective function value) is stored as an element of a Z-vector protected by an ASKFOR monitor [APPENDIX C] which prevents two processors from accessing the same node. The Z-vector elements are labeled Z(i), where i represents the level of the node in the node storage, and are not changed outside of the monitor.

STEP 4.

Any free processor (one that is not calculating a node) may ask the monitor for a simplex tableau with the maximum z-value (maximum upper bound node selection). If there is no available node, either all the nodes are fathomed and the problem is finished or some node is being calculated by some other processor and the free processor waits for the next clock tick (100ns for the HEP) and tries again. If there is an available maximum node at level i in the node storage, mark Z(i) to

indicate that the node is being processed and go to step 5.

STEP 5.

Select the first row of the node obtained from step 4 which has a non integer x variable value. This will be the row used to determine the branching variable (First Fraction Variable Selection). Add the down constraint to the simplex tableau and add the up constraint to a copy of the simplex tableau [27]. This gives the two branches (nodes) of the Branch and Bound Algorithm.

STEP 6.

Calculate the LP solution at each of these nodes by the methods of step 2. During the calculation, after each pivot operation, the processor compares the floor of the z-value of the node it is calculating with the present lower bound after each pivot operation is performed, where the floor of a number is the greatest integer less than or equal to the number. If the floor of the z-value is less than or equal to the value of the lower bound, the node is fathomed. To indicate that a node is fathomed, set Z(i)=-10E10 which will be less than or equal to the lower bound so that node will no longer be considered. Otherwise, the pivots are continued until the simplex tableau is primal feasible. For each node, if the value of the objective function Z and the associated x values are integer, Z is compared to the present lower bound and if larger, set LB=Z(i)

to indicate the new present lower bound. Whether integer or not, if the z-value is less than or equal to the present lower bound, the node is said to be fathomed. This means that no better solution will be found along that branch and hence it is pruned. When all nodes except the one with the present lower bound are fathomed, the problem is completed. If all nodes are fathomed and the lower bound is still negative infinity, there is no integer solution.

STEP 7.

If a calculated node is not fathomed it is stored in the node storage matrix. The down node will replace the branching node and the up node will form a new layer in the node storage. If the node storage has any unfathomed nodes, go to step 4; otherwise, stop and print the results.

C. Levels of Parallelism

There are several levels of parallelism possible in the implementation of the algorithm.

Level 1.

Steps 4,5,6 and 7 are combined into one logical module. Each free processor:

(a) checks the monitor for an available node and receives the node,

(b) adds the up constraint to a copy of the branching node and stores the uncalculated up node,

(c) adds the down constraint, calculates the down node, then

(d) calculates the up node, stores both results in the node storage and puts the z-values in the Z-vector.

Thus, any free processors will only have access to these nodes after both are calculated and stored.

Level 2.

The same as Level 1 except change (c) and (d) of Level 1 as follows:

(c') adds the down constraint, calculates the down node, stores it in the node storage and puts the z-value in the Z-vector, then

(d') calculates the up node, stores it in the node storage and puts the z-value in the Z-vector.

This gives the opportunity for a free processor to obtain the down node while the first processor is calculating the up node.

Level 3.

The same as level 1 except, change (b) and (d) of Level 1 as follows:

(b") adds the up constraint to a copy of the branching node and stores the uncalculated up node

in the node storage, marking it as an uncalculated node,

(d") stores the results in the node storage and puts the z-value in the Z-vector."

When the up branch is examined by a free processor it will have the Z(i) value of the node before branching. This will put it back in equal contention with all available nodes. The last line of step 4 of the algorithm would be replaced by:

If there is an available node, check to see if it is an uncalculated node. If it is, go to step 6; otherwise go to step 5.

Level 4.

Another level of parallelism is also desirable if many processors are available and the Simplex tableaus are very large. When the pivot operation is called in any of the above steps, the processor is given the indexes of the leaving row and the entering column. It then performs the element operation n*p times, where p is the number of variables and n is the sum of the number of variables, the number of constraints and the number of branches already done. These operations are independent of each other and could therefore be assigned to different processors. This assignment of processors could either be done row by row or element by element. If by row, a processor is given a row index and it performs the p operations associated with

that row. If by element, a processor is given the row and column indexes of the element it is to process.

Level 1 was used in a test version of the code on the HEP but was discarded because of its poor use of parallelism. Levels 1 and 2 are identical on a single processor but Level 2 utilizes multiple processors better because it makes the down constraint available sooner. Level 2 was implemented on the Denelcor HEP for the IBM-3 and IBM-4 test problems. There is only one line of code difference in the two levels. Level 3 was used in the simulation for all of the test problems. The simulation was done before the HEP became available and extensive changes in the code including a different node storage would have been necessary to obtain Level 3 parallelism. Level 4 could be implemented using more monitors and using methods of calculating the simplex tableaus similar to methods in the literature involving Gaussian elimination techniques [7]. This would utilize idle processors but should not change the number of pivots.

IV. SIMULATION OF A MULTIPROCESSOR

A. INTRODUCTION

A single processor was used to simulate the multiprocessor implementation of the Parallel Branch and Bound Algorithm. Level 3 parallelism was simulated using the IBM 4381 computer and the PL/I language program given in APPENDIX B. The simulation explored every node which could be applicable, which in most cases meant that every branch was pursued until either an integer lower bound was found, infeasibility occurred or the node could not be used because of its value relative to the known solution and its location in the branching tree. Three ILP techniques were studied for simulating implementation on a multiprocessor: branch and bound, branch and bound with parallel hyperplane cuts, and branch and bound with (0,1) explicit enumeration.

B. BRANCH AND BOUND

Branch and bound with first fraction variable selection and maximum upper bound node selection was used in all simulations.

The first fraction variable selection was used since it shortens the search for the branching variable in the node storage and uses only comparisons instead of calculating penalty functions.

The maximum upper bound node selection was chosen because of its simplicity, needing only a comparison search, and because of its favorable numbers of pivot calculations

when compared with the more complicated penalty function methods [28].

The Beale and Small [27] method gains storage efficiency for sequential processing, especially in the case in which the first direction contains the solution. This efficiency is gained by the use of a stack type storage with backtracking. The use of backtracking makes the variable selection automatic for one processor but more complicated for multiprocessors. The use of multiple processors also necessitates a stack for each processor and hence this method was not pursued.

C. BRANCH AND BOUND WITH PARALLEL HYPERPLANE CUTS

The techniques of part B were combined with the following addition. If a primal feasible solution does not have an integer z-value, a fractional cut is performed which is parallel to the objective function. This cut is called the "parallel hyperplane cut," and is performed immediately after a branching node is chosen and before the up and down constraints are added. This cut is different from the usual cutting plane techniques in that it is performed on the objective function row instead of a basic variable row.

D. BRANCH AND BOUND WITH EXPLICIT ENUMERATION ON SOME (0,1) VARIABLES

The calculation of the continuous solution for the first node usually takes several pivot operations, hence in the case of using a dedicated multiprocessor, all processors but one are idle during this time. Since only one

additional constraint has been added for the up or down branch, it quite frequently takes only one additional pivot to obtain the continuous feasible solution for the next node. To utilize the other processors while the first processor is calculating the first node, an explicit enumeration technique was used.

The explicit enumeration technique used gives one or more of the (0,1) variables the value 0 or 1 and then performs the calculations on the resulting simplex tableau. This tableau is smaller since at least one variable and row have been eliminated from the original tableau. An enumeration was performed by each free processor while the first processor was calculating the continuous solution for the first node. Since these processors were working on smaller simplex tableaus, it was hoped that they would have their results in fewer pivots than the first processor would need so their results would be ready for the first processor when it had completed its calculations. It was hoped that this would not only keep more processors busy while the continuous solution was being calculated, but might either give the solution or at least give a lower bound and hence a better idea of which variable to pursue next.

E. RESULTS USING THE THREE TECHNIQUES ON THE HALDI AND IBM TEST PROBLEMS.

In all implementations, the number of pivot operations was used as a measure of performance since it was found that the time to do a pivot operation did not significantly

change in a given problem as one row (up or down constraint) was added . The clock times available to this project on the IBM 4381 were in increments of ten milliseconds so were of little value.

IBM-1 through IBM-5 and HALDI-1 through HALDI-10 [1] were used as test problems in the simulation. The continuous solutions for the IBM-1, HALDI-7 and HALDI-8 were also the integer solutions so no branching was needed.

1. Branch and Bound Results. The results of the simulation of the parallel branch and bound algorithm using branch and bound techniques are given in TABLE I. Two of the twelve test problems (IBM-2 and HALDI-4) showed super linear efficiency, although the improvement was only one pivot operation less than with a single processor. Five of the test problems showed linear efficiency and five showed less than linear efficiency. Only HALDI-10 showed an increase of more than one pivot operation on the best of the multiprocessing tests over the single processor.

In general, after the optimal number of processors has been reached, the addition of more processors causes the number of nodes and the number of pivots to increase. The exceptions are HALDI-5, HALDI-6 and HALDI-9 where both the number of pivots and the number of nodes remained constant regardless of the number of processors.

With the exceptions of IBM-3 and IBM-5, no test problem would utilize more processors than the number of variables in the problem. In the case of IBM-3 using eight

TABLE I

BRANCH AND BOUND

IBM-2

7 VARIABLES

# processors	# pivots	# nodes
1	12	5
2	12	5
3	11	6
4	12	7
5	Will not use 5 processors	

IBM-3

7 VARIABLES

# processors	# pivots	# nodes
1	30	21
2	31	21
3	31	21
4	31	21
5	33	23
6	37	27
7	39	28
8	39	28
9	Will not use 9 processors	

TABLE I CONTINUED

BRANCH AND BOUND

IBM-4

15 VARIABLES

# processors	# pivots	# nodes
1	55	14
2	57	15
3	65	20
4	69	23
5	74	26
6	83	29
7	70	26
8	59	24
9	61	25
10	64	26
11	55	22
12	56	23
13	57	24
14	Will not use 14 processors	

TABLE I CONTINUED

BRANCH AND BOUND

IBM-5

15 VARIABLES

# processors	# pivots	# nodes
1	526	251
2	526	251
3	527	251
4	527	251
8	532	252

HALDI-1

5 VARIABLES

# processors	# pivots	# nodes
1	14	9
2	15	9
3	16	11
4	15	10
5	16	11
6	Will not use 6 processors	

TABLE I CONTINUED

BRANCH AND BOUND

HALDI-2

5 VARIABLES

# processors	# pivots	# nodes
1	14	9
2	15	10
3	16	11
4	15	10
5	16	11
6	Will not use 6 processors	

HALDI-3

5 VARIABLES

# processors	# pivots	# nodes
1	12	7
2	13	8
3	13	8
4	15	10
5	16	11
6	Will not use 6 processors	

TABLE I CONTINUED

BRANCH AND BOUND

HALDI-4

5 VARIABLES

# processors	# pivots	# nodes
1	14	8
2	15	9
3	13	8
4	15	10
5	16	11
6	Will not use 6 processors	

HALDI-5

5 VARIABLES

# processors	# pivots	# nodes
1	14	9
2	14	9
3	14	9
4	14	9
5	Will not use 5 processors	

TABLE I CONTINUED

BRANCH AND BOUND

HALDI-6

5 VARIABLES

# processors	# pivots	# nodes
1	11	7
2	11	7
3	Will not use 3 processors	

HALDI-9

6 VARIABLES

# processors	# pivots	# nodes
1	13	7
2	13	7
3	Will not use 3 processors	

TABLE I CONTINUED

BRANCH AND BOUND

HALDI-10

12 VARIABLES

# processors	# pivots	# nodes
1	39	9
2	43	11
3	54	14
4	53	14
5	51	17
6	57	19
7	Will not use 7 processors	

processors, the eighth processor performed only one pivot operation and following that calculation, two processors were idle. Although the simulation of IBM-5 was not carried out beyond eight processors, it appeared from the branching trees that more than fifteen processors might be utilized with very little change in the number of nodes visited, but with an increasing number of pivots.

The IBM-5 problem had almost no differences in the number of nodes visited in obtaining a solution because of the large number of distinct solutions which were at the same level of the branching tree. The slight differences in the relative number of pivots seemed to depend mostly on how many nodes were calculated with z-value floors that were the same as the z-value of the solution and on how many of their pivot operations were performed before this floor was obtained. For example, if -15 is the z-value of a feasible integer solution, whenever any pivot operation gives a value less than -14, its floor is then -15 so the node is fathomed. In the IBM-5 problem, the first feasible integer solution was the optimal solution regardless of the number of processors. Because of the enormous amount of time consumed simulating IBM-5 and the inability to see any useful patterns, further investigation of this problem was delayed until a large MIMD machine was available to check for possible patterns using parallel processors.

The IBM-4 problem seemed ideal for multiprocessors because of the path pursued using the sequential branch and

bound algorithm. The sequential algorithm pursued a path down the right branch of the branching tree where the optimal integer solution was eight levels deep with a shortest path of thirty-two pivots. There was, however, a path down the left side of the tree with an optimal solution six levels deep and with a shortest path of twenty-six pivots. A shortest path is the path a processor would take if it knew where the optimal integer solution was which would need the fewest pivot operations to obtain. In the simulation, this shortest path was not utilized until eleven processors were in use, so the advantages associated with the shortest path were offset by examining nodes which were not examined by using one processor.

2. Results Using Branch and Bound with Parallel Hyperplane Cuts. The results of this simulation are given in TABLE II. Since every node in HALDI-9 had an integer z-value it did not utilize the parallel hyperplane cuts, hence HALDI-9 is not in TABLE II. When parallel cuts were combined with the branch and bound methods, HALDI-4 was the only case where there was less than linear efficiency using the best performance of multiprocessors against single processors. Seven of the problems demonstrated super linear efficiency. HALDI-5, HALDI-6 and HALDI-9 remained at a constant level regardless of the number of processors.

No test problem, with the possible exception of IBM-5, would use as many processors as the number of variables in the problem. In HALDI-1 and HALDI-2, as the number of

TABLE II

BRANCH AND BOUND WITH PARALLEL CUTS

IBM-2

7 VARIABLES

# processors	# pivots	# nodes
1	12	4
2	11	4
3	12	5
4	will not use 4 processors	

IBM-3

7 VARIABLES

# processors	# pivots	# nodes
1	40	18
2	40	18
3	43	19
4	46	20
5	48	20
6	48	20
7	will not use 7 processors	

TABLE II CONTINUED

BRANCH AND BOUND WITH PARALLEL CUTS

IBM-4

15 VARIABLES

# processors	# pivots	# nodes	
1	90	23	
2	73	21	
3	83	24	
4	58	20	
5	54	19	
6	57	21	
7	53	20	best
8	54	21	
9	55	22	
10	56	23	
11	57	23	
12	58	24	
13	will not use 13 processors		

IBM-5

15 VARIABLES

# processors	# pivots	# nodes	
1	1137	331	
2	1098	327	
3	1133	329	
7	1081	319	best
8	1131	330	

TABLE II CONTINUED

BRANCH AND BOUND WITH PARALLEL CUTS

HALDI-1

5 VARIABLES

# processors	# pivots	# nodes
1	16	5
2	18	6
3	14	5
4	Will not use 4 processors	

HALDI-2

5 VARIABLES

# processors	# pivots	# nodes
1	19	7
2	14	5
3	13	5
4	Will not use 4 processors	

HALDI-3

5 VARIABLES

# processors	# pivots	# nodes
1	20	5
2	19	5
3	19	5
4	Will not use 4 processors	

TABLE II CONTINUED

BRANCH AND BOUND WITH PARALLEL CUTS

HALDI-4

5 VARIABLES

# processors	# pivots	# nodes
1	15	4
2	20	7
3	21	7
4	Will not use 4 processors	

HALDI-5

5 VARIABLES

# processors	# pivots	# nodes
1	21	7
2	21	7
3	21	7
4	Will not use 4 processors	

HALDI-6

5 VARIABLES

# processors	# pivots	# nodes
1	16	5
2	16	5
3	Will not use 3 processors	

TABLE II CONTINUED

BRANCH AND BOUND WITH PARALLEL CUTS

HALDI-10

5 VARIABLES

# processors	# pivots	# nodes
1	63	11
2	64	11
3	62	11
4	58	13
5	66	13
6	Will not use 6 processors	

processors increased, the efficiency increased, up to the maximum number of processors which would be used.

IBM-4 dropped from ninety pivots with a single processor to fifty-three pivots with seven processors.

These results sound very good since they show that a method is available which utilizes multiprocessors efficiently. However, these results need to be compared with the results obtained using Branch and Bound without parallel hyperplane cuts to get a more complete picture.

3. Comparison of Branch and Bound with and without Parallel Hyperplane Cuts. TABLE III gives the comparison between the two methods. In all but IBM-2 the number of pivot operations for a single processor is smaller without the parallel hyperplane cuts than with them. In IBM-2 using two processors, the number of pivot operations is the same with or without the cuts.

In IBM-5 the comparison is significant since the number of pivot operations more than doubled when the cuts were added, causing more than five hundred extra pivot operations and visiting almost one hundred extra nodes.

The comparisons in IBM-4 start out almost as bad, with fifty-five pivot operations without the cuts and ninety pivot operations with them. However, as the number of processors increases, so does the efficiency of the parallel cuts. With seven processors, the parallel cuts give better efficiency than the single processor with or without the parallel cuts. HALDI-2 is the other problem in which three

TABLE III

COMPARISONS OF THE TEST PROBLEMS USING SINGLE AND MULTIPLE PROCESSORS AND USING BRANCH AND BOUND WITH AND WITHOUT PARALLEL HYPERPLANE CUTS

Problem		Single Processor Performance	Multiple Processors Best Performance	
		Number of Pivots	Number of Pivots	Number of Processors
Haldi-1	B&B	14	15	2,4
Haldi-1	+PC	16	14	3
Haldi-2	B&B	14	15	2
Haldi-2	+PC	19	13	3
Haldi-3	B&B	12	13	2
Haldi-3	+PC	20	17	3
Haldi-4	B&B	14	13	3
Haldi-4	+PC	15	20	2
Haldi-5	B&B	14	14	2,3,4
Haldi-5	+PC	21	16	3
Haldi-6	B&B	11	11	2
Haldi-6	+PC	16	16	2
Haldi-7	B&B	The continuous solution was integer		
Haldi-8	B&B	The continuous solution was integer		
Haldi-9	B&B	13	13	2
Haldi-9	+PC	13	13	2
Haldi-10	B&B	39	43	2
Haldi-10	+PC	63	58	4
IBM-1	B&B	The continuous solution was integer		
IBM-2	B&B	12	11	3
IBM-2	+PC	11	11	2
IBM-3	B&B	30	31	2,3,4
IBM-3	+PC	40	40	2
IBM-4	B&B	55	55	11
IBM-4	+PC	90	53	7
IBM-5	B&B	526	526	2
IBM-5	+PC	1137	1081	7

B&B = Branch and Bound without Parallel Cuts
+PC = *branch and Bound with Parallel Cuts

processors with parallel cuts perform better than one processor with or without parallel cuts. Also, in HALDI-2, the number of nodes visited with three processors using the parallel cuts is less than one half the number of nodes visited with three processors not using parallel cuts.

As was stated earlier, generally fewer processors will be used than the number of variables in the problem. Also, the optimal number of processors is about one half of the number of variables. These statements are true whether parallel cuts are used or not.

4. Results Using Explicit Enumeration Techniques. The HALDI-10 test problem was chosen for solution by branch and bound using explicit enumeration, since it was the largest of the test problems which had (0,1) variables. There are twelve variables in the HALDI-10 problem and six of these are (0,1) variables. Although only five or six processors could be used in the branch and bound method with or without parallel hyperplane cuts, seven processors were used in this case. Seven processors were chosen since there were six (0,1) variables. One processor would work on the continuous solution while the other six could be working on the (0,1) variables. On these other six, one of the six variables could be set equal to one and the other five set equal to zero. This would cut the size of the resulting problems in half and perhaps give some information about feasibility together with a lower bound. It was hoped the number of pivot operations performed by the processors doing

the (0,1) enumerations would be considerably smaller than the number of pivot operations performed by the processor working on the continuous solution. However, no processor needed fewer pivot operations than the processor doing the continuous solution. The only (0,1) processor obtaining a feasible integer solution needed thirteen pivot operations while the continuous solution took only eleven. This not only gave the lower bound too late for immediate use, it also used a processor for two pivot operation periods that could have been used by one of the branches from the continuous solution. In addition, the lower bound given was too low to be of any value in the problem. The information given by the other five processors showed only that each of those solutions was feasible.

Better results were obtained by placing the (0,1) variables in the first columns, then any non-(0,1) value would be taken care of first and at a usual cost of two to four pivot operations per variable.

The results of this simulation are in TABLE IV.

TABLE IV

HALDI-10 WITH ZERO-ONE ENUMERATION

USING SEVEN PROCESSORS

Processor	Duty	# pivots	value
1	Continuous Solution	11	18.709
2	x(1)=1,Other (0,1)=0	13	12 (INT)
3	x(2)=1,Other (0,1)=0	11	14.71
4	x(3)=1,Other (0,1)=0	12	11.33
5	x(4)=1,Other (0,1)=0	12	9.11
6	x(5)=1,Other (0,1)=0	14	10.28
7	x(6)=1,Other (0,1)=0	11	10.87

OPTIMAL SOLUTION IS 17

V. IMPLEMENTATION OF PARALLEL ALGORITHM ON AN MIMD COMPUTER

A. INTRODUCTION

After the simulation was completed, the author became aware of an MIMD computer at Argonne National Laboratories which was available for graduate student research in parallel processing applications. This gave the opportunity for implementation of the Parallel Branch and Bound Algorithm on an MIMD machine.

Denelcor is the manufacturer of the machine which is called the HEP (Heterogeneous Element Processor). The HEP is described in APPENDIX A. Macros had been written to convert FORTRAN routines to routines for MIMD computers in general and the HEP in particular. Butler [4] translated some of these macros to the C language for use in his research. The adaptations of these macros for the parallel algorithm are given in APPENDIX C. The HEP had compilers for only the FORTRAN and C languages so the decision was made to convert the basic PL/I simulation code to the C language. The C language was chosen because of its ALGOL-like structure. The main problems in the basic translation were the lack of built-in functions which are so plentiful in PL/I and the subroutine structure which does not permit internal subroutines.

B. PROGRAMMING THE HEP

The early attempts to convert the single processor version of the C language program to a multiple process version met with the many frustrations of trying to think in parallel. A program may run perfectly on a single processor but when the number of processes is changed to two, the computer may abnormally end with no reasons given. The HEP architecture uses the creation of processes rather than processors since several processes may be in the pipe at the same time on any processor.

The connection to the HEP is through a modem and telephone lines. System breakdowns are frequent and the causes are not always apparent even when the system is rebooted. Since the work on the computer is usually done late at night when the phone rates are less, rebooting the computer is not always possible until the next day.

The amount of memory given to this type of project was said to be about 1.5M. With double precision arithmetic, this amount of memory is too small to run the larger test problems, hence IBM-3 and IBM-4 were chosen. IBM-3 is a good test problem to check for robustness of the code on a single processor. It is small, several of the branches lead to infeasibities and round off error can cause some of the branch and bound coding techniques to miss the optimal integer solution and instead stop with a non-optimal integer solution. IBM-3 was therefore a good candidate for checking the algorithm and the C language code on the parallel

process computer. IBM-4 was chosen because of the extensive simulation effort already done. The simulation had shown that a large number of ties (nodes with the same z-values) were encountered causing a large fluctuation in the paths taken to a solution. There are also several possible solutions.

The results from the HEP are printed in the order in which the information gets to the front-end-machine which handles the I/O and the control program. All processes compete for the I/O buffers. The results are not separated on the printout in terms of processes. The simulation results help to place results with processes.

Locks were used around all print statements. Without them, if two processes want to print at the same time, the printout is garbled with intermixed messages. Since locks slow the machine, fewer and shorter print statements were designed. The change in the size of print statements can have an effect on which node a process examines.

Global variables are needed so all the processes can have access to shared information. The Z-vector was protected by a monitor and not changed outside of it. The code, given in APPENDIX D, has the access to this ASKFOR monitor on line 123. The finding of the next node to be examined is done through the GETPROB macro which is contained in the monitor.

The heart of the parallelism is the "work" subroutine. All other subroutines needed in the parallelism are called

from this module. The part of the PL/I simulation program that was the main program has been replaced by lines 90 through 98. These lines call the macros which CREATE the number of processes, time the parallelism, start the problem, call the "work" module and end the parallel part of the program.

C. RESULTS OF THE TEST PROBLEMS IMPLEMENTED ON THE HEP

Both IBM-3 and IBM-4 were run successfully as single processor problems on the VAX-11/780 at Rolla before transferring the code to the HEP. The problems were then run with "numprocs" (number of processes) set equal to one. More processes were then added to see whether the actual runs would agree with the simulation. Since the parallel processes code is written for level 2 parallelism, the original branching trees [27] were used to check the results. These results are given in TABLE V and correspond with the Level 2 simulation results. Clock times in units of 100ns were also given for the problems after the continuous solution was calculated and problems were being assigned to the multiple processors.

IBM-3 was run with one through six processes with very little difference in the numbers of pivots from the results obtained in the simulation. The times for parallelism indicated that two processes complete the problem in about one half the time of one processor. However, with three processes the number of pivots is larger and the time is not close to being as small as one third of the time for one

TABLE V

IMPLEMENTATION OF LEVEL 2 PARALLELISM ON HEP

IBM-3

# processors	# pivots	time in ns
1	30	250432900
2	30	139863000
3	31	123222300
4	33	119084700
5	34	120791500
6	37	131717100

IBM-4

# processors	# pivots	time in ns
1	55	979241500
2	123	1407789100
3	128	1073274500
4	154	1063886900
6	155	836087200
7	134	691446400
8	143	693102900

processor. As the number of processes increase from four to five, the time actually increases by .001 sec, and the number of pivots increase from thirty-three to thirty-four. An increase of three pivots in going from five to six processes increases the time by .01 sec.

The IBM-4 results indicate a vast difference between implementation on the HEP using Level 2 parallelism and the simulation which used Level 3 parallelism (compare TABLE I with TABLE V). Level 2 causes the free processes to wait until the down node is calculated before the up node can be obtained for calculation, even though the up row had been added and the up node stored before the down row was added. Level 2 parallelism causes each process to act on more complete information before obtaining a node to process than level 3 parallelism does. This takes time, and in the case of IBM-4, caused the calculation of many more nodes than with level 3 parallelism.

Because of the difficulty involved in getting on the HEP and in staying on it for long enough periods of time to do multiple runs, IBM-4 was run using one, two, three, four, six, seven and eight processes. Another factor was, the IBM-4 problem used enough more memory with multiple processes (since each process is allotted its own memory), that sometimes memory exceptions would occur merely by changing the number of processes. Sometimes this would terminate the session.

In going from one process to four, the number of pivots almost tripled while the processing time only went up slightly. With four processes, one of the processes (or combinations of processes) took the shortest path to a solution, but this was not enough to make up for all of the nodes calculated by the other three processes. With one process the shortest path to a solution was of length thirty-two, hence only twenty-three pivots were performed on other nodes, nineteen of which were performed to obtain the continuous solution. With three and four processes the shortest path to their solution was of length twenty-six, using the other pivots for other nodes (each of the solutions will have a shortest path).

With two and seven processes, the shortest path to their solution was of length thirty-five. In this case, two and one half times as many processes took half as much time to do only eleven more pivots. This shows that, not only was more work done with more processes but also, the average time per process per pivot increased. Six and eight processes took distinct paths with shortest path length of thirty four. The eight different numbers of processes found five different solutions.

VI. A CASE STUDY

A. THE SYSTEM DESIGN PROBLEM

The following system design (SD) problem is described by Plane and McMillan [29].

An electric power company plans to build a steam generating plant capable of producing 2 million kilowatt hours of electrical energy per day. The major equipment in the plant will consist of boilers, generators, and condensers. The sources of supply for these pieces of equipment have been narrowed to 11 manufacturers. In TABLE VI are presented data relative to the equipment offered by these suppliers (A through K).

The power company wants to design that system which will meet the energy capacity of the plant with the least cost. The requirements are a follows:

A. The capacity of the set of generators selected must be at least 2 million kwh/day.

B. The steam requirements of the generators must be met by the combined capacities of the boilers selected.

C. The steam capacities of the set of condensers selected must be adequate to accomodate the steam capacities of the boilers.

D. Equipment of one supplier is interchangeable with that of another supplier except in the case of supplier A. Note that supplier A produces both boilers and condensers. The operating costs quoted for supplier A's boiler and condenser

TABLE VI

EQUIPMENT OFFERED BY SUPPLIERS FOR THE SYSTEM DESIGN MODEL

		Steam Kcfm (capacity)	Electricity Kwh/day	Initial Cost ($K)	Operating Cost $K/yr
Boilers					
X(1)	A	100		50	50
X(2)	B	140		75	60
X(3)	C	90		60	40
X(4)	D	80		50	20
Generators (requirements)					
X(5)	E	70	500	600	60
X(6)	F	120	650	600	75
X(7)	G	150	700	800	75
X(8)	H	100	800	750	90
Condensers (capacity)					
X(9)	A	50		25	3
X(10)	I	65		17	4
X(11)	J	70		20	4
X(12)	K	55		13	2

are based on the assumption that each of A's boilers will be matched with two of A's condensers, since they constitute a matched set. If each and every one of A's boilers is not used with two of A's condensers then an added $10,000 annual operating cost can be expected.

E. Because of their size and shape it is impracticable to fit one of supplier F's generators into the plant with one or more of supplier G's generators.

F. Supplier K's condensers are of such construction that they must be used in pairs.

G. Management has decided the initial capital outlay for the system should not exceed $2.3 million.

Assume the power company's objective is to minimize the expected annual operating cost of the system, subject to the above constraints. This leads to the following integer linear programming formulation.

$$\text{MAXIMIZE } -50X(1)-60X(2)-40X(3)-20X(4)-60X(5)-75X(6)-75X(7)$$
$$-90X(8)-3X(9)-4X(10)-4X(11)-4X(12)-10X(13)$$

SUBJECT TO

$$500X(5)+650X(6)+700X(7)+800X(8) \geq 2000$$
$$100X(1)+140X(2)+90X(3)+80X(4)$$
$$-70X(5)-120X(6)-150X(7)-100X(8) \geq 0$$
$$50X(9)+65X(10)+70X(11)+110X(12) \geq 100X(1)+140X(2)$$
$$+90X(3)+80X(4)$$
$$2X(1)-X(9) \leq 100X(13)$$
$$X(6) \leq 4-4X(14)$$
$$X(7) \leq 3-3(1-X(14))$$

$$50X(1)+75X(2)+60X(3)+50X(4)$$
$$+600X(5)+600X(6)+800X(7)$$
$$+750X(8)+25X(9)+17X(10)$$
$$+20X(11)+26X(12) \leq 2300$$

$X(i) \geq 0$, $X(i)$ integer, and $X(13), X(14) = (0,1)$

This problem is notorious for poor performance using cutting plane methods, especially the all integer cuts. Syslo [30], did not obtain a solution after 15 minutes of computation time on an Amdahl 470 V/6 and 350,000 iterations.

B. APPLICATION OF THE ALGORITHM

The SD problem was attempted using the VAX 11/780 with the dimensions of the node storage matrix declared as 50,16,50, where the first dimension is the number of rows anticipated in the simplex tableau, the second is the number of columns and the third is the number of layers needed in the node storage. These were larger than any of the test problems except IBM-5. However, the problem did not run. The first and third dimensions were then increased to 100, each with similar results. The problem was then partially simulated on the IBM 4381 to find what to expect for an upper bound on the dimensions. The number of layers of the node storage was found to be at least 117 and the number of rows needed to get there was determined to be no more than 35. With new dimensions of 35,15,125, the problem obtained the solution in 501 pivots with a depth of 10 in the branching tree and a shortest path to the only solution

obtained was of length 21.

This problem was larger than any of the IBM problems tried on the HEP. When the problem was attempted on the HEP, the error message indicated a memory problem and the program abnormally terminated. The next run brought the system down. The first attempt to take care of the problem was to call Argonne and ask for more memory, however it was not clear to the people there how this could be done and the Denelcor representative had just been changed. They did not think that it was a memory allocation problem.

The next attempt was to split the problem into two problems and test it on the UMR VAX, eliminating X(6) on one run and X(7) on the next. This would also eliminate X(14). In eliminating X(6), 124 levels of the node storage were needed, 424 pivots were performed and all exposed nodes were fathomed with no integer solution. The elimination of X(7) proved more fruitful, giving the solution in 312 pivots, using 87 levels of node storage. The splitting, however took a total of 736 pivots with at least 624 needed if two processes communicated with each other to know when the solution had been found. Also, the size of the smaller subproblem was still too large to obtain a solution on the HEP if lack of allocated memory was the problem.

By cutting the dimensions down to 33,15,40, the HEP performed until these limits were reached with multiple processes. However, the algorithm did not get close to a solution, but verified that the code would run if the memory

problem was corrected or more memory could be allocated.

The HEP, however, was not the problem. After system breakdowns at intervals of approximately two hours for almost a week, the problem was found to be in the way the VAX front-end-machine operating system was looking at the memory. The operating system was executing code which should never have been executed. Possibly a memory error was sending the code to the wrong place and thus causing the system breakdowns. New memory boards were installed and the system immediately went down. By hiding 4M of the original memory from the operating system and allowing it access only to the 8M of newly installed memory the system breakdowns stopped. Even though this memory problem was on the VAX and not the HEP, this temporary fix allowed a continuation of the study of the SD problem, although the memory problem with the front-end-machine has not been resolved. The current logon messages indicate a need to save results immediately since system breakdowns are still occurring daily.

The use of MIMD machines with their front-end and the associated operating system problems are still not fully understood.

C. RESULTS USING THE HEP

Times in terms of 100ns intervals were taken for the parallel portion of the code (APPENDIX C, lines 94-97). Speedup is defined to be the quotient of the time for a single process to complete a task and the time for n

processes to complete the same task. For n processes, linear speed-up would be n. If the quotient is greater than n, super linear speed-up is said to be obtained. Efficiency is as defined in CHAPTER IV.

The following results can be seen from TABLE VII. A speedup in time using two processes is 1.97 and linear efficiency is obtained. With three processes, a speed-up of 2.86 and super linear efficiency is obtained. It was not until eight processes were in use that the super linear efficiency becomes significant where only 495 pivots were necessary. With 8,10 and 16 processes, successive runs sometimes gave a different number of pivots. This occurred because with that many processes, completion of a pivot and the reporting of the results to the Z-vector may occur in the same clock time (100ns interval) on two processes and therefore the same path to the solution may not always be followed . In these cases, the times varied slightly also, hence the times and number of pivots for these runs for a given number of processes were averaged.

D. CONCLUSIONS

As a general rule in the test problems, more processes than the number of variables should not be utilized. However, in the SD problem which has 14 variables, not only were 16 processes utilized, but the efficiency generally increased as the number of processes increased. This indicates that in some real life type models, super linear

TABLE VII

SYSTEM DESIGN MODEL RESULTS
USING THE PARALLEL BRANCH AND BOUND ALGORITHM
AND THE HEP MULTIPROCESSOR

Number of Processors	Average Number of Pivots	Average Number Seconds in Parallel Processing
1	501	12.034
2	501	6.115
3	500	4.215
4	503	3.307
5	504	2.845
6	506	2.482
7	500	2.195
8	495	1.982
9	494	1.839
10	496	1.848
12	495	1.830
14	481	1.762
16	472	1.793

efficiency can be obtained and the number of processes efficiently utilized may be more than was indicated by the test problems.

VII. CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

A. CONCLUSIONS

The use of parallel processing in the solution of general integer linear programming models is desirable in some cases as is evidenced by the study of the System Design model and the Haldi and IBM test problems.

Super linear efficiency is obtainable for some types of integer linear programming problems using parallel branch and bound techniques and the simplex method. In the System Design problem, as the number of processes was increased, better efficiency was achieved up to the maximum number of processes available. For the Haldi and IBM test problems, TABLE I shows that the best efficiency was obtained using approximately one half as many processes as the number of variables.

In general, if multiple processes give linear or super linear efficiency, there exists a point at which the addition of more processes degrade the performance. One reason for this is that with multiple processes, nodes are explored that would not have been explored with a single process. This can be seen in TABLE I and TABLE II by noting the number of nodes examined as the number of processes is increased. A way to overcome this is for multiple processes to pursue a shorter path to a solution. This may mean more nodes, but it must mean fewer pivots.

TABLE I and TABLE II also show that there is an upper limit for the number of processes which can be utilized. There can never be fewer pivots performed by a process than the length of the shortest path to a solution. When some process, or combination of processes, uses this path, any additional processes added will not improve the efficiency. On the other hand, if the branch and bound algorithm using a single process causes a path other than the shortest one to be pursued, multiple processes may give better efficiency.

For single processors, the addition of parallel cuts generally did not improve the efficiency. In some problems, it more than doubled the number of pivots needed. Using parallel cuts generally started out inefficiently but improved as the number of processes was increased. The combination of parallel hyperplane cuts with branch and bound increased efficiency in only one fourth of the test problems.

The use of explicit enumeration in conjunction with branch and bound for the purpose of keeping processes busy does not seem to be efficient. A much better way to keep the extra processes busy while one process is finding the continuous solution would be to use level 4 parallelism.

The present code is portable between MIMD machines except for a few lines in some of the macros. The code is robust, giving correct answers to the System Design model and to the Haldi and IBM problems tested for which the allotted memory space allowed the program to finish.

The algorithm is general and leaves choices as to which path to follow in the branching tree. One such choice is which node should be chosen when there is a tie, that is whenever two nodes have the same z-value. This became evident when level 2 parallelism was used on IBM-4. For nodes with the same z-value, level 2 parallelism, combined with the storage numbering method of the program, caused the choosing of the smallest numbered node, whereas the simulation using level 3 parallelism used the first-in-first-out technique for choosing the node.

B. SUGGESTIONS FOR FUTURE RESEARCH

Brown and Almasi [31] believe that a new era in high performance computing is beginning which will be dominated by parallel computing and that its application will pace future development in manufacturing and knowledge-intensive industries.

The research started in this paper is now being continued in collaboration with Ralph Butler. Our research is centered on the implementation of level 4 parallelism. As more memory becomes available for this type of research on the HEP or other MIMD type computers, larger problems could be used to test the algorithm.

Different techniques for handling the node storage could be tried which would use a different numbering system on the nodes. This may give a better way to tell which process is working with a particular node and make the tracing of shortest paths easier. It might also give more

efficient use of the node storage, by eliminating a level from the storage as soon as the node is fathomed and reusing it. Some different method of keeping track of the best present lower bound would also need to be devised.

Techniques involving a combination of parallel hyperplane cuts with branch and bound may still have merit if used in a different way. An example might be to use the cuts only when the branch and bound technique gets the same z-value for two or three consecutive pivots, or nodes.

Parallelism should be investigated using new techniques for solving linear programming problems. Extensions to the C language like those suggested by Nacini [32] could also be studied.

BIBLIOGRAPHY

[1] Garfinkel, R. S. and Nemhauser, G. L., Integer Programming, John Wiley and Sons, New York, (1972).

[2] Lusk, E. L. and Overbeek, R. A., "Use of Monitors in Fortran: A Tutorial on the Barrier, Self Scheduling Do-Loop, and ASKFOR Monitors," ANL-84-51, Argonne National Laboratory, Argonne,IL, (July 1984).

[3] Lusk, E. L. and Overbeek, R. A., Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors," Technical Report ANL-83-97, Argonne National Laboratory, Argonne,IL, (July 1984).

[4] Butler, R. A., "An Algorithm for Parallel Subsumption,"Unpublished Ph.D. dissertation, University of Missouri-Rolla, (May 1985).

[5] Hwang, K. and Briggs, F. A., Computer Architecture and Parallel Processing, McGraw-Hill, New York, (1984).

[6] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, 20, 4, (1978), p 740-776.

[7] Lord, R. E., Kowalik, J. S., and Kumar, S. P., "Solving Linear Algebraic Equations on am MIMD Computer," Journal of the ACM, 30, 1, (1983), p 103-117.

[8] Daniel, R. C.,"LP-Based Mathematical Programming--The Significance of Recent Developments," The Journal of the Operational Research Society, 32, 2 (1981), p113-118.

[9] Kumar, S. P. and Kowalik, J. S., " Parallel Factorization of a Positive Definite Matrix on an MIMD Computer," Proceedings of the 1984 International Conference on Parallel Processing, Computer Society Press, (1984).

[10] Fuller, S. J.,Ousterhout, J. K., Raskin, L., Rubinfeld, P. I., Sindhu, P. J. and Swan, R. J., "Multi-Microprocessors: An Overview and Working Example," Proceedings of the IEEE, 66,2,(1978), p216-228.

[11] Gehringer, E. F., Jones, A. K., and Segall, Z. Z., "The Cm* Testbed," Computer, 15, 10, (1982), p 40-53.

[12] Sameh, A. H., "Numerical Parallel Algorithms--A Survey," High Speed Computer and Algorithm Organization, Academic Press, New York, (1977), p207-228.

[13] Traub, J. F., "Iterative Solution of Tridiagonal Systems on Parallel or Vector Computers," Complexity for Sequential and Parallel Numerical Algorithms, Academic Press, New York, (1973) p49-82.

[14] Quinn, M. J. and Yoo, Y. B. "Data Structures for the Efficient Solution of Graph Theoretic Problems on Tightly-Coupled MIMD Computers", Proceedings of the 1984 International Conference on Parallel Processing, Computer Society Press, (1984).

[15] Witt, B. I., "Parallelism, Pipelines, and Partitions: Variations on Communication Modules," Computer, 18, 2, (1985), p105-112.

[16] Barlow, R. H., "Performance Measures for Parallel Algorithms," Parallel Processing Systems, an Advanced Course, (Evans,D.E.,Editor), Cambridge: Cambridge University Press, (1982), p179-189.

[17] Jones, A. and Schwarz, R., "Experience Using Multiprocessor Systems- A Status Report", Computing Surveys, 12, 2, (1980), p121-165.

[18] Andrews, G. R. and Schdeider, F. B. "Concepts and Notations for Concurrent Programming", ACM Computing Surveys, 15, 1, (1983).

[19] Deitel, H. M., An Introduction to Operating Systems, Addison Wesley Publishing Company, Reading,MS, (1984).

[20] Lai, T. H., "Anomalies in Parallel Branch and Bound Algorithms," Proceedings of the 1983 International Conference on Parallel Processing, IEEE Computer Society Press, (1983), p183-190.

[21] Satyanarayanan, M. Multiprocessors: A Comparative Study, Prentice Hall, Englewood Cliffs,NJ, (1980).

[22] Li, G. and Wah, B. W., "How To Cope With Anomalies In Parallel Approximate Branch-And-Bound Algorithms," AAAI-84 National Conference of Artificial Intelligence, (August 1984)

[23] Kumar, V. and Kanal, L., "Parallel Branch-and-Bound Formulations for AND/OR Tree Search," Department of Computer Sciences University of Texas at Austin, TR-83-14, Austin,Texas, (August 1983).

[24] Quinn, M. J. and Deo, N., "An Upper Bound for the Speedup of Parallel Branch-and-Bound Algorithms," Computer Science Department, Washington State University, CS-83-112, Pullman, Washington, (May 1983).

[25] Deo, N., Yoo, T. B. and Lord, R. E., "A Parallel Algorithm for the One-to-All, Mixed-Weight, Shortest Path Problem," Computer Science Department, Washington State University, CS-83-113, Pullman, Washington, (June 1983).

[26] Gillett, B. E., Introduction to Operations Research, A Computer-Oriented Algorithmic Approach, McGraw Hill Book Company, New York, (1976).

[27] Taha, H. A.,Integer Programming Theory, Applications, and Computations, Academic Press, New York, (1975).

[28] Chambless, S. D., "Users Guide to an Integer Programming Package," Unpublished Paper, Computer Science Department, University of Missouri, Rolla, (May 1984).

[29] Plane, D. R. and McMillan, C., Discrete Optimization, Integer Programming and Network Analysis for Management Decisions, Prentice Hall, Englewood Cliffs, NJ, (1971), p153,159.

[30] Syslo, M. M., Deo, N. and Kowalik, J. S., Discrete Optimization Algorithms with Pascal Programs, Prentice Hall, Englewood Cliffs, NJ, (1983), p94,96.

[31] Brown, J. C. and Almasi, G., "Research in Parallel Computing," Computer, 17, 7, (1984), p92-93.

[32] Nacini, R.,"A Few Statement Types Adapt C-language to Parallel Processing," Electronics, 57, 13, (1984), p125-129.

[33] Smith, B. J., "A Pipelined, Shared Resource MIMD Computer," Proceedings of the International Conference on Parallel Processing, Computer Science Press, (August 1978).

[34] Mullin, R., Nemeth, E. and Weidenhofer, N., "Will Public Key Crypto Systems Live Up To Their Expectations?" Proceedings of the 1984 International Conference on Parallel Processing, Computer Society Press, (1984).

[35] Heterogeneous Element Processor (HEP) Hardware Reference Manual, Publication 9000003, Denelcor,Inc., (1982).

VITA

Rochelle Lloyd Boehning was born on November 12, 1932 near Diamond, Missouri. He received his elementary and secondary schooling in Seneca, Missouri. He received his college education from Joplin Junior College in Joplin, Missouri; Northeastern Oklahoma A & M in Miami, Oklahoma; Kansas State College of Pittsburg in Pittsburg, Kansas; the University of Missouri-Columbia, in Columbia, Missouri; the University of Kansas in Lawrence, Kansas; the University of Wisconsin-Madison, in Madison, Wisconsin; the Illinois Institute of Technology, in Chicago, Illinois; the University of Arkansas, in Fayetteville, Arkansas; and the University of Missouri-Rolla in Rolla, Missouri. He received a Bachelor of Science in Education degree in Mathematics in July, 1959 and a Master of Science degree in Mathematics in July, 1960 from Kansas State College of Pittsburg, in Pittsburg Kansas. He received a Master of Science degree in Computer Science in July, 1983 from the University of Missouri-Rolla in Rolla, Missouri.

He has been enrolled in the Graduate School of the University of Missouri-Rolla since August, 1982.

APPENDIX A

THE DENELCOR HEP (HETEROGENEOUS ELEMENT PROCESSOR)

The HEP is a large-scale, high-speed, general-purpose mainframe data processor. It is designed for applications that can effectively use a processing speed of 10 to 160 million instructions per second (Mips). HEP achieves this throughput with a multiple instruction stream, multiple data stream (MIMD) architecture.

MIMD architecture allows user processes, or programs, to execute in parallel. Each process has its own independent instruction stream operating on its own data stream. Processes cooperate by sharing data and solving parts of the same problem in parallel. In HEP, high-use logic functions are pipelined to further increase performance so that new inputs can start processing without waiting for previous input to finish. Also, all of the instruction streams and their active data streams are always in main memory; even though processes share the computing resource, no active time is required to load and store processes when selected to run.

The hardware components that make the HEP system unique are the central processing unit (CPU), the switch module, and the data memory module.

The CPU is the basic computing unit of the HEP system. There are as many as 16 CPUs in a machine configuration.

Each CPU includes 2K 64-bit words of register memory, 4K 64-bit words of constant memory, at least 32K 64-bit words of program memory, 64 user processes, 64 supervisor processes, nine function execution units, MIMD architecture and pipelined logic.

A process in a HEP CPU is an instruction stream (or program) stored in program memory for execution. To be executed, a process must be created in an active task.

A task is the fundamental protection domain in a CPU. When a task is activated, explicit areas of each type of memory are defined for process use in that task. Tasks can overlap if they are to cooperate in solving a common problem. Processes can be created and terminated in a task whenever appropriate to optimize parallelism and maximize throughput.

In a CPU, instruction processing is available uniformly to all active tasks. Each 100 nanoseconds (ns), a task is selected and one instruction from a process in the task is accepted for processing. The processes for a task are queued so that only one instruction for a process is accessible at a time. Instructions are processed on a first-in, first-out sequence according to the position of the process in the task queue.

When a process is read out of a task queue, it enters a control pipeline known as the instruction loop, where decoding and operand fetching are performed. The instruction loop is divided into eight 100ns time phases. An instruction takes 800ns to get through the pipeline but

there can be eight instructions in the pipeline at once.

The function units, such as float adder, float/integer multiplier, and hardware access unit, are all completed in 800ns in synchronism with the instruction loop. Functions which are not completed in 800ns, such as the divider and the scheduler function unit (SFU), are called asynchronous functions. The divide operation takes 1700ns to complete. To maintain the throughput, the divider is replicated rather that pipelined and can accept new operands and begin execution every 100ns until all modules are busy. Process data synchronization is maintained by inhibiting access to the memory location receiving the result of the asynchronous function until after the result has been stored.

The SFU controls all operations that access data memory. Asynchronous SFU operations require a random, intermediate time to complete, so the SFU withholds the process from the task queue and maintains it in an SFU queue. When complete, the process is requeued using a special asynchronous access port.

The HEP switch is a flexibly-configured, programmable network that interconnects CPUs, data memory modules, I/O control processors (a VAX 11/780 in our case) and other system devices. It uses packet switching techniques to route messages among the units that comprise the system. Each node in the switch network has three full-duplex ports, so it can simultaneously send and receive three messages. Each message processed by the switch contains the address of

the unit to which it is directed and the data being transmitted. Each message has an associated age (priority), which is incremented each time the message is routed if the original routing is other than the optimal direction. Each node has an input rate of 100ns. The propagation time through the node is 50ns. Therefore, the switch is configured in such a way that adjacent nodes have alternate input cycles.

Data memory provides communication and process synchronization between tasks active in different CPUs. All data memory in HEP occupies one continuous address space, regardless of the number of memory units. The entire memory is addressable by all the CPUs via the switch. Local data memory can be made available to each CPU by special allocation if needed . SFU access to data memory via the switch is asynchronous with the instruction and data loops. The SFU uses control logic to synchronize multiple concurrent accesses to data memory and to ensure correct relinking to the task queues when the data memory operation is complete. Synchronous and asynchronous access to data memory are through separate ports, so accessing conflicts do not occur.

Program memory stores instruction streams to be executed. Because of the execute-only characteristics of program memory and the ability to subdivide other memory domains by indexed addressing, data environments need not be bound to instruction streams. Several processes can

execute the same instruction stream using separate data streams in other memories. Conflict between writing synchronous and asynchronous results is avoided by using separate access times to the memories [2,3,5,33,34,35].

APPENDIX B

PL/I PROGRAM TO SIMULATE BRANCH AND BOUND TECHNIQUES WITH AND WITHOUT PARALLEL HYPERPLANE CUTS

```
//C9040D JOB (0465VS1B,LPGM,C9040D,UMRVMB),'BOEHNING,CHELLE
//  TIME=1,MSGCLASS=A
//    EXEC  PLIXCLG,PARM.PLI='GOSTMT,MARGINS(2,72,1)'
//PLI.SYSPRINT DD DUMMY
//PLI.SYSIN DD *
 LEXDUAL: PROC OPTIONS(MAIN);
 DCL  A(0:99,0:31)
      FLOAT DEC(16) INIT((3100)0),
      (MINC,SUM,MINCOST,MINACT,SUM1,FK,QUO)FLOAT DEC(16),
      (E,I,J,K,L,M,N,P,Q,R,MINCOL,MINRW,FLAG,FLAG1,COL_DN,
      COL_UP)FIXED DEC(3),T CHAR(9),
      (MILLI,T1,T2,ST1,ST2,LT2,ELAPSED_TIME,TOTAL_TIME,
      LOAD_TIME)FIXED DEC(5),
      (SYSIN,SYSPRINT)FILE,
      (ABS,SUBSTR,FLOOR,CEIL,MOD,TIME,CHAR)BUILTIN;
     /*************************************************/
     /* LOAD REQUIREMENTS=B(I) AND ACTIVITIES= A(I,J) */
     /*************************************************/
    GET LIST(P,Q,R);/* P=NO. OF VARIABLES, Q=NO. OF
    CONSTRAINTS*/
    DO I=P+1 TO R;
         GET LIST((A(I,J) DO J=0 TO P));
```

```
END;
DO I=P+1 TO R;
   DO J=0 TO P;
      A(I,J)=-1*A(I,J)
   END;
END;
 /*****************************/
 /* LOAD NON BASIC VARIABLES */
 /*****************************/
DO J=1 TO P;
     A(J,J)= -1;
END;
 /***********************************/
 /* LOAD COST COEFFICIENTS= C(J) */
 /***********************************/
DO J=1 TO P;
     GET LIST(A(0,J));
     A(0,J)=(-1*A(0,J));
END;
CALL MINCOLM;
N=R;
IF FLAG=1 THEN DO;  /*DUAL INFEASABLE*/
     N=R+1;
     CALL NEWROW;
     L=N;
     E=MINCOL;
     CALL SIMPLEX;
```

```
      CALL PIVOT; /*THE SIMPLEX IS NOW DUAL FEASABLE*/
      END;
 DO UNTIL (MINRW=0);
 CALL ROUNDA;
 CALL MINROW;
   IF MINRW=0
   THEN DO;            /*PRIMAL FEASIBILITY*/
           L=0;
           E=0;
           CALL PRNTSOL;
         END;
   ELSE DO;
           L=MINRW;
           CALL LEXMIN;
           IF FLAG1=0
              THEN DO;
                   E=0;
                   CALL SIMPLEX;
                   PUT SKIP(3) LIST('NO FEASABLE
                                     SOLUTION');
                   MINRW=0;
                   END;
              ELSE
                   CALL PIVOT;
      END;
   END;
DO M=1 TO 20 UNTIL (FK=0);
```

```
IF(((A(0,0)-FLOOR(A(0,0)))>1E-10)
  &((CEIL(A(0,0))-A(0,0)>1E-10)))
THEN DO;
        CALL ROUNDA;
        CALL PCUT;PUT SKIP LIST('PARALLEL CUT');
        L=N;
        CALL LEXMIN;
        CALL PIVOT;
        DO UNTIL(MINRW=0);
          CALL MINROW;
          IF MINRW=0
          THEN DO;
                  L=0; E=0;
                  CALL PRNTSOL;
                END;
          ELSE DO;
                  L=MINRW;
                  CALL LEXMIN;
                  IF FLAG1=0
                  THEN DO;
                    E=0;
                    CALL SIMPLEX;
                    PUT SKIP (3) LIST('NO FEASIBLE
                                       SOLUTION');
                    MINRW=0;
                    END;
                 ELSE CALL PIVOT;
```

```
            END;
        END;
  END;
ELSE PUT SKIP(3) LIST('Z IS INTEGER ');
CALL ROUNDA;
CALL FIRSTFRACTION;
IF FK=0
THEN
  DO;
    PUT SKIP LIST('INTEGER SOLUTION');
    CALL PRNTSOL;
  END;
ELSE
  DO;
   IF  M=2 : M=3
   THEN
    DO;
    CALL UPBRANCH; PUT SKIP LIST('UP BRANCH');
    IF COL_UP=0
    THEN PUT SKIP LIST('UP   BRANCH IS INFEASIBLE');
    ELSE
      DO;
        E=COL_UP;
        L=N;
      END;
    END;
   ELSE
```

```
DO;
CALL DNBRANCH; PUT SKIP LIST('DOWN BRANCH');
IF COL_DN=0
THEN PUT SKIP LIST('DOWN BRANCH IS INFEASIBLE');
ELSE
  DO;
    E=COL_DN;
    L=N;
  END;
END;
    CALL PIVOT;
    DO UNTIL(MINRW=0);
     CALL MINROW;
     IF MINRW=0
     THEN DO;
             L=0; E=0;
             CALL PRNTSOL;
           END;
     ELSE DO;
             L=MINRW;
             CALL LEXMIN;
             IF FLAG1=0
             THEN DO;
               E=0;
               CALL SIMPLEX;
               PUT SKIP (3) LIST('NO FEASIBLE
                                  SOLUTION');
```

```
                    MINRW=0;
                    END;
                  ELSE CALL PIVOT;
                END;
            END;
        END;
     END;
    PUT SKIP EDIT('M=',M)(A,F(3));
    /*************************************************/
    /*  THIS GIVES THE CONTINUOUS SOLUTION          */
    /*************************************************/
    /*********************************************/
    /*                NEWROW                     */
    /*LOAD THE ROW TO OBTAIN DUAL FEASABILITY*/
    /*********************************************/
NEWROW:PROC;
  SUM=0;
  DO J=1 TO P;
   SUM=SUM+ABS(A(0,J));
   A(N,J)=1;
  END;
    SUM1=0;
  DO I=1 TO R;
    SUM1=SUM1+ABS(A(I,0));
  END;
    IF SUM1>SUM*10
    THEN A(N,0)=SUM1;
```

```
     ELSE  A(N,0)=SUM*10;
   END NEWROW;
   /*****************************************/
   /*                MINCOLM                */
   /* FINDS MOST NEGATIVE COST COEFFICIENT */
   /*****************************************/
MINCOLM: PROC;
   FLAG=0;
   MINCOST=-.0001;
   MINCOL=0;
   DO J = 1 TO P;
      IF MINCOST>A(0,J)
         THEN DO;
              FLAG=1;
              MINCOST=A(0,J);
              MINCOL=J;
              END;
   END;
END MINCOLM;
              /*****************************/
              /* FINDS MOST NEGATIVE ROW */
              /*****************************/
MINROW: PROC;
   MINACT=-.00001;
   MINRW=0;
   DO I=1 TO N;
      IF A(I,0)<-.00001
```

```
      THEN
       IF MINACT> A(I,0)
       THEN DO;
            MINACT= A(I,0);
            MINRW = I;
            END;
   END;
END MINROW;
                /*****************************/
                /*   FINDS THE MIN COLUMN    */
                /*****************************/
LEXMIN: PROC;
    MINC=10E11;
    FLAG1=0;
    DO J= 1 TO P;
      IF A(L,J)<-.0001
         THEN IF
              MINC>ABS(A(0,J)/A(L,J))
              THEN DO;
                   FLAG1=1;
                   MINC=ABS(A(0,J)/A(L,J));
                   E=J;
                   END;
     END;
END LEXMIN;
                /**********************/
                /* CALCULATES A(I,J) */
```

```
                /**********************/
PIVOT: PROC;
   T=TIME; MILLI=SUBSTR(T,5,5);T1=MILLI;
   DO I=0 TO N;
     IF I^=L THEN
      DO J=0 TO P;
        IF J^=E THEN
                A(I,J)=A(I,E)*A(L,J)/(-1*A(L,E))+A(I,J);
      END;
   END;
   DO I=0 TO N;
      IF I^=L THEN
      A(I,E)=A(I,E)/(-1*A(L,E));
   END;
   DO J=0 TO P;
      IF J^=E THEN A(L,J)=0;
      ELSE A(L,E)=-1;
   END;
   T=TIME; MILLI=SUBSTR(T,5,5);T2=MILLI;
   IF T2>=T1 THEN
    ELAPSED_TIME=T2-T1;
   ELSE DO T2=T2+60;
        ELAPSED_TIME=T2-T1;END;
   PUT SKIP(2) EDIT('ELAPSED TIME =',ELAPSED_TIME)(A,F(5));
   PUT SKIP EDIT('Z=',A(0,0))(A,F(16,8));
 END PIVOT;
```

```
                /************************/
                /* PRINTS THE SIMPLEX */
                /************************/
SIMPLEX: PROC;
   /* L IS LEAVING VARIABLE, E IS ENTERING COLUMN*/
   PUT SKIP(3) EDIT('L=',L)(COL(2),A,F(3));
   PUT SKIP(2) EDIT('E=',E)(COL(2),A,F(3));
   DO I= 0 TO N;
      PUT SKIP(2) LIST(' ');
      PUT EDIT((A(I,J) DO J=0 TO P))(F(9,1));
   END;
   PUT SKIP(3) LIST('********************************');
   END SIMPLEX;
                /*********************/
                /* PRINTS SOLUTION */
                /*********************/
PRNTSOL: PROC;
   DO I=0 TO P;
      IF I=0 THEN
         PUT SKIP(2) EDIT('OPTIMAL VALUE',A(0,0))
                        (COL(2),A,F(12,5));
      ELSE PUT SKIP(2) EDIT('X',CHAR(I),'=',A(I,0))
                        (COL(2),3 A,F(12,5));
    END;
 END PRNTSOL;
```

```
   /*****************************************/
   /* ROUNDS TO INTEGERS IF THE VALUES ARE  */
   /*      WITHIN E-10 OF AN INTEGER        */
   /*****************************************/
 ROUNDA: PROC;
   DO I=0 TO N;
     DO J=0 TO P;
       IF (A(I,J)-FLOOR(A(I,J)))<1E-10
       THEN A(I,J)=FLOOR(A(I,J));
       ELSE IF (CEIL(A(I,J))-A(I,J))<1E-10
            THEN A(I,J)=CEIL(A(I,J));
      END;
    END;
   END ROUNDA;
                /*******************************/
                /*  PARALLEL CUTTING PLANE     */
                /*  CUTS Z TO INTEGER VALUE    */
                /*******************************/
PCUT: PROC;
   K=0;FK=0;
    DO I=0 TO P UNTIL (FK^=0);
       FK=A(I,0)-FLOOR(A(I,0));
       K=I;
    END;
    IF FK=0
    THEN   PUT SKIP(3) LIST('INTEGER SOLUTION');
```

```
    ELSE DO;
            N=N+1;
            A(N,0)=-FK;
              DO J=1 TO P;
                A(N,J)=FLOOR(A(K,J))-A(K,J);
              END;
           END;
END PCUT;
    /***********************************/
    /*  FINDS FIRST FRACTIONAL VALUE */
    /*        IN THE FIRST COLUMN     */
    /***********************************/
FIRSTFRACTION: PROC;
  FK=0; K=0;
  DO I=1 TO P UNTIL(FK^=0);
    FK=A(I,0)-FLOOR(A(I,0));
    IF FK^=0
    THEN
      DO;
        K=I;
        PUT SKIP EDIT('FK=',FK,'K=',K)(A,F(5,2),A,F(3));
      END;
  END;
END FIRSTFRACTION;
     /*****************************************/
     /*  ADDS DOWN BRANCH IF IT IS FEASIBLE  */
     /*****************************************/
```

```
DNBRANCH: PROC;
  QUO=1E10; COL_DN=0;
  DO J=1 TO P;
    IF A(K,J)>0
    THEN
      IF QUO>A(0,J)/A(K,J)
      THEN
        DO;
          QUO=A(0,J)/A(K,J);
          COL_DN=J;
        END;
   END;
   IF COL_DN^=0
   THEN
     DO;
       N=N+1;
       A(N,0)=-FK;
       DO J=1 TO P;
         A(N,J)=-A(K,J);
       END;
     END;
 END DNBRANCH;
     /*****************************************/
     /*  ADDS UP BRANCH IF IT IS FEASIBLE     */
     /*****************************************/
UPBRANCH: PROC;
  QUO=1E10; COL_UP=0;
```

```
    DO J=1 TO P;
      IF A(K,J)<0
      THEN
        IF QUO>ABS(A(0,J)/A(K,J))
        THEN
          DO;
            QUO=ABS(A(0,J)/A(K,J));
            COL_UP=J;
          END;
     END;
     IF COL_UP^=0
     THEN
       DO;
         N=N+1;
         A(N,0)=FK-1;
         DO J=1 TO P;
           A(N,J)=A(K,J);
         END;
       END;
   END UPBRANCH;
   PUT SKIP(2) LIST('IBM6 PARALLEL CUTS WITH B AND B');
END LEXDUAL;
//LKED.SYSPRINT DD DUMMY
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSIN DD *

/*
```

APPENDIX C

MACROS USED IN THE C PROGRAM

```
          /*******  macro definitions  *********/
#define GETPROB(D1,D2) \    /* USED IN THE ASKFOR MONITOR
        if (m > -1) \           TO GET THE MAXIMUM UPPER
           { \                  BOUND VALUE AND SUBSCRIPT */
             MAXZ(); \
             D1 = br; \
             D2 = 0; \
            }


#define RESET  \
        m = -1;


#define PROBSTRT \
         MENTER(s1,0) \
         m = 0; \
         CONTINUE(s1,0,0) \
         MEXIT(s1,0)
```

```
#define MENTER(D1,D2)          /* ENTER THE MONITOR */
#define CONTINUE(D1,D2,D3)
#define MEXIT(D1,D2)           /* LEAVE THE MONITOR */
#define NEWPROC(D1)            /* START NEW PROCESS*/
#define AINIT(D1)        /* INITIALIZE THE ASKFOR MONITOR */
#define ADEC(D1)         /* DECLARE THE ASKFOR MONITOR */
#define BARINIT(D1)      /* INITIALIZE THE BARRIER MONITOR*/
#define LOCKINIT(D1)     /* INITIALIZE THE LOCKS */
#define LOCKDEC(D1)      /* DECLARE THE LOCKS */
#define LOCK(D1)         /* USED FÓR PRINT STATEMENTS */
#define UNLOCK(D1)
#define BARRIER(D1,D2)   /* DEFINED BY LUSK & OVERBEEK */
#define PROBEND(D1,D2)   /* END OF PROBLEM */
#define PROGEND(D1)      /* END OF PROGRAM */
#define CREATE(D1)       /* CREATES PROCESSES */
#define CLOCK(D1) D1=0;
#define ASKFOR(D1,D2,D3,D4,D5) D2=1; D4
                         /* DEFINED BY LUSK AND OVERBEEK */
```

APPENDIX D

C LANGUAGE PROGRAM FOR THE PARALLEL BRANCH AND BOUND ALGORITHM WITH MACROS, FOR THE DENELCOR HEP, ARGONNE NATIONAL LABORATORY

```
1.  #include<stdio.h>
2.  #include<math.h>
3.  int N[20],br,rc,m,lb,clk1,clk2,pct;
4.  int p,q,r,numprocs;
5.  double B[50][16][20],Z[20],C,LB;
6.  NEWPROC(slave)
7.  ADEC(sl)
8.  LOCKDEC(3)
9.  main()
10. {
11.     /* load requirements b(i) and activities a(i,j)*/
12. int i,j,flag,mincol,sum,suml;
13. double mincost,ffchek();
14. AINIT(sl)
15. LOCKINIT(3)
16.
17. scanf("%d",&numprocs);
18. printf(" numprocs = %d\n",numprocs);
19. scanf("%d %d %d",&p,&q,&r);
20. printf("p= %d   q= %d   r= %d\n",p,q,r);
21.
```

```
22.  for(i=p+1;i<r+1;i++){
23.     for(j=0;j<p+1;j++)
24.     {
25.        scanf("%f",&B[i][j][0]);
26.     }
27.  }
28.   /* load non basic variables */
29.  for(j=1;j<p+1;j++){
30.     B[j][j][0]=(-1);
31.  }
32.
33.   /* load cost coefficients c(j) */
34.
35.  for(j=1;j<p+1;j++){
36.     scanf("%f",&B[0][j][0]);
37.     B[0][j][0]=(-1)*B[0][j][0];
38.  }
39.  m=0;
40.    /* find most negative cost coefficient */
41.  pct=0;
42.  flag=0;
43.  mincost=(-.0001);
44.  mincol=0;
45.  for(j=1;j<p+1;j++){
46.     if(mincost>B[0][j][0])
47.     {
```

```
48.         flag=1;
49.         mincost=B[0][j][0];
50.         mincol=j;
51.       }
52.   }
53.   if(flag==1){          /* dual infeasible */
54.     r=r+1;
55.    /* add a new row of 1's */
56.     sum=0;
57.     for(j=1;j<=p;j++){
58.         sum=sum+fabs(B[0][j][0]);
59.         B[r][j][0]=1;
60.     }
61.     sum1=0;
62.     for(i=1;i<=r;i++){
63.         sum1=sum1+fabs(B[i][0][0]);
64.     }
65.     if(sum1>sum*10){
66.       B[r][0][0]=sum1;
67.     }
68.     else{
69.       B[r][0][0]=sum*10;
70.     }
71.     pivot(r, m,r,mincol);
72.     printsol(m);
73. } /* The simplex is now dual feasible */
74.    /* Now we try for primal feasibility */
```

```
75. LB=(-10e10);
76. primal(r,m);
77. /* Uses Primal to obtain the primal solution */
78. C=Z[0]=B[0][0][0];
79. N[0]=r;
80. printf("CONTINUOUS SOLUTION IS %f\n",C);
81. /*****************************************/
82. /* This Gives The Continuous Solution */
83. /*****************************************/
84. if(ffchek(m)==0){
85.    printf("CONTINUOUS SOLUTION IS INTEGER\n");
86.    lb=0;
87.    goto answer;
88. }
89. /******************/
90. RESET
91. for (i=1; i < numprocs; i++) {
92.    CREATE(slave);
93. }
94. CLOCK(clk1)
95. PROBSTRT
96. work ('m');
97. CLOCK(clk2)
98. PROGEND(s1)
99. /******************/
100. answer:
```

```
101.  printf("pivot count =%d\n",pct);
102.  printf("\nm=%d\n",m);
103.  printf("INTEGER SOLUTION Z[%d]=%f\n",lb,B[0][0][lb]);
104.  printf(" \n total time was %d\n", clk2 - clk1);
105.  printf("lb=%d   LB=%f\n",lb,LB);
106.  printsol(lb);
107. } /* end main */
108./*********************************************************/
109./*                  subroutines                          */
110./*********************************************************/
111./*********************************************************/
112.    /*****************************/
113.   slave () { work ('s'); }
114.   /*****************************/
115./*********************************************************/
116.   work (who)
117./*********************************************************/
118.   char who;
119.   {
120.   int i, j, mw,n, brw, arc;
121.   double ffchek();
122.   for (;;) {    /* forever */
123.      ASKFOR(sl,arc,numprocs,GETPROB(brw,arc),RESET)
124.      if (arc == -1  || (arc != 0 && who == 'm'))
125.         break;
126.      if (arc != 0) continue;
```

```
127.      if(pct>=100 || m>=19){    /* safety valve */
128.         LOCK(1)
129.         printf("pivot count=%d\n",pct);
130.         UNLOCK(1)
131.         break;
132.      }
133.   /******************/
134.      n=N[brw];
135.      dnbrn(n, brw);          /* adds the down row */
136.      if(dnbrn(n,brw)==1) printf("error\n");
137.      MENTER(s1,0)
138.      m=m+1; mw=m;
139.      Z[m]=(-10e5);
140.      MEXIT(s1,0)
141.      for(i=0;i<=n;i++){
142.         for(j=0;j<=p;j++)
143.         B[i][j][mw]=B[i][j][brw]; /* copies node */
145.      }
146.      upbrn(n,mw);              /* adds the up row */
147.      N[brw]=N[mw]=n+1;
148.      if(Z[brw]<(-10e6)) goto nosol1;
149.      n=N[brw];
150.      primal(n,brw);   /* calculates the down node */
151.  /*********************************/
152.      if(B[0][0][brw]>LB){
153.       if(ffchek(brw)==0){/*check for integer vector*/
154.          if(B[0][0][brw]!=floor(C)){
```

```
155.              LOCK(1)
156.              printf("NEW INTEGER LOWER BOUND,
                  Zd[%d]=%f\n",brw,B[0][0][brw]);
157.              UNLOCK(1)
158.              if(LB<=(-10e5) || B[0][0][brw]>=LB){
159.                LB=B[0][0][brw];
160.                MENTER(s1,0)
161.                Z[brw]=(-10e10);
162.                MEXIT(s1,0)
163.                lb=brw;
164.                for(i=0;i<=mw;i++){
165.                   if(LB<floor(Z[i])) goto nosoll;
166.                }
167.              }
168.            }
169.            lb=brw;
170.            if (who == 'm') goto endwork;
171.          }
172.        }
173.        MENTER(s1,0)
174.        Z[brw]=B[0][0][brw];
175.        MEXIT(s1,0)
176.    nosoll:
177.    /*************************************************/
178.    if(Z[mw]<(-10e6)) goto nosolu;
179.    n=N[mw];
180.    primal(n,mw);          /* calculate the up node */
```

```
181.   /*****************************************************/
182.   if(B[0][0][mw]>LB){
183.     if(ffchek(mw)==0){/* check for integer vector */
184.       if(B[0][0][mw]!=floor(C)){
185.          LOCK(1)
186.          printf("NEW INTEGER LOWER BOUND,
           Zu[%d]=%f\n",mw,B[0][0][mw]);
186.          UNLOCK(1)
187.          if(LB<=(-10e5) :: B[0][0][mw]>=LB){
188.             LB=B[0][0][mw];
189.             Z[mw]=(-10e10);
190.             lb=mw;
191.             for(i=0;i<=mw;i++){
192.                if(LB<floor(Z[i])) goto nosolu;
193.             }
194.          }
195.          lb=mw;
196.          if (who == 'm') goto endwork;
197.       }
198.     }
199.  /*****************************************************/
200.     MENTER(s1,0)
201.     Z[mw]=B[0][0][mw];
202.     MEXIT(s1,0)
203.  nosolu:printf("");
204.  } /* end forever */
205.  endwork:return (0); } /*end work */
```

```
/*******************************************************/
MAXZ()/* Calculates the present Maximum Upper Bound*/
/*******************************************************/
{
int i,frc;
double MAX;
br=(-1);
rc=(-1);
frc=0;
MAX=LB;
for(i=0;i<=m;i++){
   if(Z[i]<=LB && Z[i] > (-10e4)){
      Z[i]=(-10e10); /*NODE FATHOMED*/
      printf("node Z[%d] fathomed \n",i);
      continue;
   }
   if(Z[i]>MAX && Z[i]>(-10e4)){
     MAX=Z[i];
     br=i;
     rc=0;
   }
   if(Z[i]>(-10e6) && Z[i]<(-10e4)) frc=1;
}  /* end for */
if(rc==0) Z[br]=(-10e5);
else if(frc==1) rc=1;
return(0);
} /* end MAXZ */
```

```
233./*****************************************************/
234./*****************************************************/
235.primal(n, x)  /* Uses Pivot and Lexmin
                    to obtain the primal solution */
236./*****************************************************/
237. int n, x;
238. {
239. int L, E;
240. do{
241.    L=minrow(n, x);
242.    if(L!=0){
243.      E=lexmin(x,L);
244.      if(E!=0){
245.         pivot(n,x,L,E);
246.         LOCK(1)
247.         printf("Z[%d]=%f\n",x,B[0][0][x]);
248.         UNLOCK(1)
249.         if(floor(B[0][0][x])<=LB){
250.            L=0;
251.            MENTER(s1,0)
252.            Z[x]=B[0][0][x]=(-10e10);
253.            MEXIT(s1,0)
254.            LOCK(1)
255.            printf("NODE Z[%d] FATHOMED\n",x);
256.            UNLOCK(1)
257.         }
258.         else printf("");
```

```
259.      }
260.      else{
261.         MENTER(s1,0)
262.         Z[x]=B[0][0][x]=(-10e10);
263.         MEXIT(s1,0)
264.         LOCK(1)
265.         printf("No Feasible Solution for Z[%d]\n",x);
266.         UNLOCK(1)
267.         L=0;
268.      }
269.    }
270. }
271. while(L!=0);
272. } /* End Primal */
273. /*****************************************************/
274.  pivot(n, x, L, E)
275  /* Performs the pivot operation on the Simplex    */
276. /*****************************************************/
277. int n, x, L, E;
278. {
279. int i, j;
280. pct=pct+1;
281. for(i=0; i <= n; i++) {
282.    if (i != L){
283.       for (j=0; j<=p; j++) {
284.          if (j != E){
```

```
285.              B[i][j][x]=B[i][E][x]*B[L][j][x]
                  /((-1)*B[L][E][x])+B[i][j][x];
286.            }
287.         }
288.      }
289. }
290. for(i=0;i<=n;i++){
291.      if(i!=L){
292.        (B[i][E][x]=B[i][E][x]/((-1)*B[L][E][x]));
293.      }
294, }
295. for(j=0;j<=p;j++){
296.      if(j!=E)(B[L][j][x]=0);
297.      else(B[L][E][x]=(-1));
298. }
299. return(0);
300. } /* end pivot*/
301. /*******************************************/
302.  minrow(n, x) /* Finds the most neg. row */
303. /*******************************************/
304.  int n, x;
305.  {
306.  int i,minrw;
307.  double minact;
308.  minact=(-.00001);
309.  minrw=0;
310.  for(i=1;i<=n;i++){
```

```
311.      if(B[i][0][x]<(-.00001)){
312.        if(B[i][0][x]<minact){
313.           minact=B[i][0][x];
314.           minrw=i;
315.        }
316.      }
317.   }
318.   return(minrw);
319.   } /* end minrow */
320.   /*******************************************/
321.   lexmin(x,L) /*  Finds the minimum column  */
322.   /*******************************************/
323.   int x,L;
324.   {
325.   double minc;
326.   int j,E;
327.   minc=10e11;
328.   E=0;
329.   for(j=1;j<=p;j++){
330.      if(B[L][j][x]<(-.00001)){
331.         if(minc>fabs(B[0][j][x]/B[L][j][x])){
332.            minc=fabs(B[0][j][x]/B[L][j][x]);
333.            E=j;
334.         }
335.      }
336.   }
337.  return(E);
```

```
} /* end lexmin */
/************************************************/
double ffchek(x)
/*Checks for an integer lower bound */
/************************************************/
int x;
{
int i;
double ff;
for(i=1;i<=p;i++){
    ff=(B[i][0][x])-floor(B[i][0][x]);
    if(ff>0.000000001 && ff<0.999999999) break;
}
if(ff<=0.000000001 || ff>=0.999999999) ff=0.0;
return(ff);
} /* end ffchek */
/************************************************/
dnbrn(n, x) /*Adds Down Branch if it is Feasible*/
/************************************************/
int n, x;
{
int i,j,k,coldn;
double quo,fk;
for(i=1;i<=p;i++){
    fk=(B[i][0][x])-floor(B[i][0][x]);
    if(fk>0.000000001 && fk<0.999999999){
       k=i;
```

```
365.        break;
366.     }
367. }
368. if(fk<=0.000000001 || fk>=0.999999999) return(1);
369. quo=1e10;
370. coldn=0;
371. for(j=1;j<=p;j++){
372.     if(B[k][j][x]>0){
373.       if(quo>(B[0][j][x]/B[k][j][x])){
374.          quo=B[0][j][x]/B[k][j][x];
375.          coldn=j;
376.       }
377.     }
378. }
379. if(coldn!=0){
380.    n++;
381.    B[n][0][x]=(-fk);
382.    for(j=1;j<=p;j++){
383.       B[n][j][x]=(-B[k][j][x]);
384.    }
385. }
386. else{
387.    MENTER(s1,0)
388.    Z[x]=(-10e10);
389.    MEXIT(s1,0)
390. } return(0);
391. } /*end downbranch*/
```

```
392. /*************************************************/
393.  upbrn(n, x) /*Adds Up Branch if it is Feasible*/
394. /*************************************************/
395. int n, x;
396. {
397. double quo,fk;
398. int i,j,k,colup;
399. for(i=1;i<=p;i++){
400.   fk=(B[i][0][x])-floor(B[i][0][x]);
401.   if(fk>0.000000001 && fk<0.999999999){
402.     k=i;
403.     break;
404.   }
405. }
406. quo=1e10;
407. colup=0;
408. for(j=1;j<=p;j++){
409.    if(B[k][j][x]<0){
410.      if (quo > fabs(B[0][j][x] / B[k][j][x])){
411.         quo = fabs(B[0][j][x] / B[k][j][x]);
412.         colup = j;
413.      }
414.    }
415. }
416. if(colup!=0){
417.   n++;
418.   B[n][0][x]=fk-1;
```

```
419.    for(j=1;j<=p;j++){
420.       B[n][j][x]=B[k][j][x];
421.    }
422. }
423. else{
424.    MENTER(s1,0)
425.    Z[x]=(-10e10);
426.    MEXIT(s1,0)
427. }
428. return(0);
429. } /* End Up Branch */
430. /*****************************************************/
431. printsol(x)  /* Prints the solution */
432. /*****************************************************/
433. int x;
434. {
435. int i,j;
436. printf("\n");
437. for(i=0;i<p+1;i++){
438. printf("\n");
439.     printf("x(%d)= %f",i,B[i][0][x]);
440. }
441. printf("\n");
442. printf("\n");
443. } /* end printsol */
```