Very Large-scale Linear Programming: A Case Study in Exploiting Both Parallelism and Distributed Memory

Anne Kilgore

CRPC-TR93354-S
December 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

RICE UNIVERSITY

VERY LARGE-SCALE LINEAR PROGRAMMING: A CASE STUDY IN EXPLOITING BOTH PARALLELISM AND DISTRIBUTED MEMORY

by

Anne Kilgore

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Arts

APPROVED, THESIS COMMITTEE:

Robert E. Bixby, Chairman
Professor of Computational and Applied Mathematics

Virginia Torczon
Research Scientist in Computational and Applied Mathematics

John E. Dennis, Jr.
Noah Harding Professor of Computational and Applied Mathematics

---

Robert Michael Lewis
Research Scientist in Computational and Applied Mathematics

Linda Torczon
Faculty Fellow in the Department of Computer Science

Houston, Texas
December, 1993

Abstract

VERY LARGE-SCALE LINEAR PROGRAMMING: A CASE STUDY IN EXPLOITING BOTH PARALLELISM AND DISTRIBUTED MEMORY

by

Anne Kilgore

There has been limited success with parallel implementations of both the simplex method and interior point methods for solving real-world linear programs. Experience with a parallel implementation of CPLEX, a state of the art implementation of the simplex method, on an Intel distributed-memory multiprocessor machine will be described. We will exploit the structure of the class of problems arising from airline crew scheduling. A particular instance with 12,753,313 variables will be studied. This instance is too large to fit on current sequential machines in standard linear programming data structures. We will show how our implementation exploits both distributed memory and parallelism and allows the full problem to be kept in memory. Finally, we will discuss algorithmic ideas that our implementation affords us and show results for a variant of the *greatest decrease* algorithm, an idea suggested many years ago but never tested on linear programming problems of significant size.

Acknowledgments

First, I would like to thank my parents for their support. I owe a special thank you to my husband, Scott, for first encouraging me to return to school.

I would like to thank the members of my committee. I owe special thanks to John Dennis who first suggested that I take on this research and urged me to stay. Robert Bixby has shared his extensive knowledge of both CPLEX and linear programming which has proven invaluable. Michael Lewis has shared not only his computer expertise, but also an occasional remark that would quickly put everything back into perspective. Linda Torczon has carefully commented on this work, adding both clarity and conciseness.

I owe a very special thank you to my advisor, Virginia Torczon. She has stayed with me throughout the project, always keeping me on track, and always with an optimistic attitude. She has spent hours teaching me about parallel computation, and in general teaching me the viewpoint of a mathematician. A better advisor would be hard to find.

I would also like to thank the many people at Rice who have shared their knowledge with me. I would like to thank Richard Tapia for his help and encouragement in the beginning. A very special thanks to Michael Pearlman for creating my data sets and managing 530 megabytes of data, as well as sharing his unlimited knowledge of Unix. Thanks to Mark Messina for his help with the Intel machines.

I also wish to acknowledge the Caltech support group for their very professional and friendly attitude during my use of their computers.

I need to thank Irv Lustig for the wonderful routines from his work on the Cray Y-MP. Also thanks to John Gregory of Cray Research for retrieving the 530 megabyte data file.

I would also like to thank Michael Trosset for his statistical expertise that allowed us to generate an appropriate set of test problems, and also for his support and enthusiasm.

Some important people I wish to thank are my fellow students. To Rudy Elizondo who beat me to graduation but with whom I shared many long hours. To Wei Zuo whose support from the beginning continued throughout the years. Also a very big thank you to Natalia Alexandrov, Shireen Dadmehr, Tony Kearsley, and Eva Lee whose support, caring, and humor helped improve each day.

And last but not least, there is Morgan, whose smile was always there to greet me at the end of the day.

Contents

Illustrations

Tables

Chapter 1

Introduction

The evolution of parallel computers has encouraged the development of new algorithms. The implementation of many of these new algorithms on current sequential computers would be impractical as they would not run efficiently. However, the development of parallel computers, along with recent improvements in implementations of the simplex method, have motivated new interest in solving large-scale linear programs. We will discuss a new parallel implementation of the simplex algorithm for solving a particular instance of a restricted class of these large-scale linear programs on a distributed-memory machine. We will use CPLEX,* an implementation by R.E. Bixby of the simplex method [3]. We will show how it is possible to exploit both parallelism and distributed memory to solve linear programs more efficiently when the ratio of the number of columns to the number of rows is disproportionately large.

The current generation of distributed-memory machines typically have a small amount of memory per processor. However, when a large number of processors are used—512 processors for our work—the collective memory is large enough to allow problems with millions of columns to be completely loaded into memory. The parallelism allows the work load to be distributed, thereby decreasing the amount of time spent per iteration. This combination of parallelism and distributed memory has the additional advantage of allowing all of the columns to be examined at every iteration and then choosing, by some criterion, the "best" column. By using the primal simplex method we are able to test a variant of the *greatest decrease* criterion as the

*CPLEX is a trademark of CPLEX Optimization, Inc.

rule for the selection of a new basis at each iteration. Our discuss will focus on the characteristics of this new parallel implementation of an existing algorithm.

1.1 The linear program

The particular class of problems we will study were generated by American Airlines. They are large set partitioning models arising from airline crew scheduling problems. An airline company must schedule its crews to cover each flight leg, where a flight leg is a nonstop flight between two cities. Each of these flight legs is part of a round-trip flight schedule, referred to as a "pairing." Each pairing must start at the home base of a crew and return to that home base. Each pairing must conform not only to company policy and available staff, but also to FAA regulations. Each row of the constraint matrix represents a flight leg to be flown and each column represents a legal round-trip pairing.

In general, airlines have different sets of crews qualified to work with specific types of equipment. The scheduling models can be broken down into subsets that cover a certain type of equipment, say a Boeing 747 airplane. As an example, consider an airline that wishes to schedule its Boeing 747s and assume that the company has 100 airplanes of this type. That particular set of planes will perform a certain number of take-offs and landings (flight legs) per day. It has been shown [1] that the growth in the number of columns generated is exponential in the number of rows (flight legs) of the constraint matrix. In addition, the airline needs to schedule on a monthly basis. Thus, the amount of data involved is a very important consideration. Fortunately, the daily problems can be solved first and these solutions can be used for the weekly problems. The solutions for the weekly problems can be used to obtain solutions for the monthly problems. However, while the number of columns can be reduced somewhat, the size of the daily problem still remains very large. For this reason the solution to the daily problem is achieved by solving a succession of smaller subproblems [1].

The task of formulating this class of problems requires the generation of the allowed pairings, called *pairing generation*, or in more general terms referred to as *column generation*. This procedure can involve as much computational work as solving the actual set partitioning problem [1]. In column generation, duplicate pairings are generated, though these can be removed easily. Also, many of the generated pairings, even though they are legal pairings, would never be used. In this case it would be beneficial not to generate these pairings in the first place. However, it is very difficult to identify such pairings. While the problem of efficiently generating the crew pairings is an interesting one, and much work has been done to improve the process, column generation is not the focus of this work.

A set partitioning problem can be formulated as an Integer Program (IP) as shown in equation (1.1). The technique most often used for solving an IP is to solve a sequence of linear programming relaxations in order to solve, or attempt to solve, the IP. In this process the integrality constraint is first relaxed. The resulting linear program (LP) is then solved. The solution to the LP is used to generate new constraints for the IP in an effort to force an integer solution. The process is then repeated until an optimal integer solution is found or it can be shown that no such solution exists. The IP of the set partitioning problem is

$$\begin{aligned} min \;\; & c^T x \\ s.t. \; & Ax = b \\ & x \in \{0,1\}^n, \end{aligned} \tag{1.1}$$

where $A \in \{0,1\}^{m \times n}, b = (1,1,\ldots,1)^T \in \Re^m$, and $c \in \Re^n$. Assuming no row of A is all zero, the corresponding linear programming relaxation takes the form

$$\begin{aligned} min \;\; & c^T x \\ s.t. \; & Ax = b \\ & x \geq 0, \end{aligned} \tag{1.2}$$

where $A \in \{0,1\}^{m \times n}, b = (1,1,\ldots,1)^T \in \Re^m$, and $x, c \in \Re^n$. For the crew scheduling problem, the constraint matrix A consists of $0,1$ entries such that

$$a_{ij} = \begin{cases} 1 & \text{if flight leg } i \text{ is contained in pairing } j \\ 0 & \text{otherwise} \end{cases}$$

and the variables have the interpretation

$$x_j = \begin{cases} 1 & \text{if pairing } j \text{ is selected to be flown} \\ 0 & \text{otherwise.} \end{cases}$$

Once a particular column (crew pairing) is chosen, the flight legs covered by that crew pairing cannot be covered by another crew pairing. This is enforced by the requirement $Ax = b$, where $b = (1,1,\ldots,1)^T$.

A particular instance of this set partitioning problem, which we refer to as the American Airlines Challenge Problem, has 837 rows (flight legs) and 12,753,313 columns (legal round-trip pairings). The constraint matrix A, with at most eighteen nonzero entries per column, has just under one hundred million (99,845,826) nonzero entries. The ascii file containing the row indices for each column of A consumes on the order of one-half gigabyte of disk space. The size of the Challenge problem is a driving force for this research. In particular, we intend to exploit the disproportionate ratio of columns to rows.

1.2 Motivation

There are three key considerations that motivate this work.

- The sheer size of the Challenge problem leads to very large memory requirements. The data file used to generate the test set used in our work consumes 530 megabytes of disk space.
- The disproportionate number of columns to rows means that, for the primal simplex method, the computation of the reduced costs accounts for up to 99% of the total execution time (ignoring the time needed to initialize the problem).

- Our interest in developing new implementations of optimization algorithms that take advantage of parallel architectures.

For these reasons we think that our investigation is helpful in learning more about distributed memory, parallelism, and linear programming.

1.2.1 Memory requirements

Even for the computers available today, 530 megabytes of data—the amount of data required to define the Challenge problem—is difficult to handle. For this reason, techniques that reference only portions of the data at any one time have been developed for handling these large data sets. John Forrest suggested a particular method, referred to as *Sprint*, [7] for handling this situation in airline crew scheduling. Bixby, Gregory, Lustig, Marsten, and Shanno [2] implemented this method, which they called *sifting*, on a CrayY-MP[†].

The desire to avoid the need to read and write columns to disk during the optimization phase was the key to our choice of the Intel Touchstone Delta[‡] machine. With sixteen megabytes of memory per node, and a total of 512 nodes, aggregate memory is 8192 megabytes. Consequently, even after factoring in overhead for the operating system and CPLEX, we are able to load all of the data into memory, thus eliminating any disk I/O after the initial setup.

1.2.2 Work load

In order to exploit the disproportionate ratio of columns to rows while using the primal form of the simplex method, we break the nonbasic columns into subsets and distribute these subsets among the processors. Each processor will compute the

[†]Cray and Cray Y-MP are trademarks of Cray Research, Inc.

[‡]Touchstone Delta is a trademark of Intel Corporation.

reduced costs for its subset in parallel, thus distributing the work load and thereby substantially reducing the amount of time spent in each iteration.

Most of the work for this particular class of problems, up to 99% when executed on a single-processor machine, is in the computation of the reduced costs and related calculations. For this computation, we must first solve an $m \times m$ linear system to obtain the shadow prices and then calculate the actual reduced costs by performing $n-m$ inner products. When $n \gg m$ and n is very large this is a non-trivial calculation. Hence it is the reduced costs, and not the forward and backward solves, that create the large work load.

1.2.3 New parallel implementations

The principal benefit of our parallel implementation is that we are able to partition the calculation of the reduced costs, thus reducing the time spent per iteration. In addition, we are able to use our parallel implementation to test a variant of *greatest decrease* in the objective function value as the criterion for selecting an entering column. It has been conjectured that this selection criterion would give the best progress. However, to obtain the objective function value for each possible entering column, a large portion of the work for a simplex iteration would have to be completed once for each of these columns. The extra work makes implementing the greatest decrease criterion seem impractical in a sequential computing environment.

It is the desire to test greatest decrease that has determined our choice of the primal simplex method. If we were to use the dual simplex method we could also distribute the data and the work load in an analogous manner. However, we would not have been able to test the greatest decrease in the objective function as a selection criterion for an entering column, at least not in the same natural way. By using the primal simplex method we can exploit both distributed memory and parallelism and also test a variant of greatest decrease.

In our parallel implementation of the simplex method, each processor computes the reduced costs for its subset of nonbasic columns and selects an entering and leaving pair of columns. As part of this process, the change in the objective function that would occur if this pair were chosen is calculated. Thus, every processor has, for its subset of nonbasic columns, candidates for entering and leaving the basis that can now be represented by the objective function change. At this stage each processor has a candidate pair and its associated objective function change. We can compare the objective function change for each pair of columns, across the processors, and choose from among these objective function changes the associated column that would give the most decrease. So given p processors we can compute, with little overhead, the actual change in the objective function value for up to p possible choices of entering columns. It is important to notice that as the number of processors changes so does the number of possible choices of objective function changes. There will be a different choice of iterates depending on the number of processors used and hence the algorithm changes with the number of processors. In this respect, by changing the selection criterion for the entering column we can change the algorithm.

1.3 Summary

In this investigation we will show how to use the simplex method to take advantage of the structure of the Challenge problem. We will show how to exploit both distributed memory and parallelism by distributing both the 12.75 million columns and the work load among the processors, thus reducing the time spent per iteration. With this parallel implementation, using the primal formulation of the simplex method, we will show results for a variant of greatest decrease used as the selection criterion for an entering column.

Chapter 2

Algorithm

The linear program we wish to solve is:

$$\begin{aligned} min \quad & c^T x \\ s.t. \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.1}$$

where $A \in \{0,1\}^{m \times n}, b = (1,1\ldots,1)^T \in \Re^m$, and $x, c \in \Re^n$.

Let us reorder the columns of the constraint matrix A and the objective coefficients c so that the first m correspond to a feasible basis (B), with the remaining columns being nonbasic (N). Assuming a feasible basis exists, this gives

$$\begin{aligned} min \quad & c_B^T x_B + c_N^T x_N \\ s.t. \quad & A_B x_B + A_N x_N = b \\ & x_B, x_N \geq 0. \end{aligned}$$

To make the steps of the simplex algorithm clearer, we denote the objective function value $c^T x$ by z and the change in the objective function value by Δz. We can write the basic variables x_B and z in terms of the nonbasic variables x_N.

$$\begin{aligned} x_B &= A_B^{-1} b - A_B^{-1} A_N x_N \\ z &= c_B^T A_B^{-1} b + (c_N^T - c_B^T A_B^{-1} A_N) x_N. \end{aligned}$$

We now present a formal statement of the revised simplex algorithm.

2.1 The revised simplex algorithm

Given an initial basis A_B,

while $d_j \ngeq 0 \ \forall \ j \in N$ **do**

Step 1: Solve $\pi^T A_B = c_B^T$. /* compute the shadow prices

Step 2: Compute $d_N^T = c_N^T - \pi^T A_N$. /* compute the reduced costs

if $d_j \geq 0 \ \forall \ j \in N$, **then** we are optimal.

else choose $j_e \in N$ such that $d_{j_e} < 0$.

Step 3: Solve $A_B y = A_{j_e}$. /* compute y in terms of the basis

Step 4: Compute Θ. /* compute the step length

Step 5: Update the basis.

Go to *Step 1.*

2.2 A description of the revised simplex algorithm

We will briefly review the five steps of the revised simplex method and emphasize the areas that are important for our parallel implementation.

2.2.1 Step 1. Calculate the shadow prices.

Let $\pi^T = c_B^T A_B^{-1}$. To find π we solve the linear system

$$\pi^T A_B = c_B^T.$$

Before optimality the entries of π are referred to as the shadow prices. At optimality they constitute a feasible dual solution.

2.2.2 Step 2. Calculate the reduced costs.

Next we calculate the reduced costs of the nonbasic variables:

$$d_N^T = c_N^T - \pi^T A_N.$$

If we find

$$d_j \geq 0$$

for all $j \in N$, then x is an optimal solution; otherwise, we choose $j_e \in N$ such that $d_{j_e} < 0$.

This is the stage in our implementation that gives us the flexibility to modify the criterion we use to select the entering variable. The choice of a "good" entering variable is one of the key questions we will address.

2.2.3 Step 3. Solve for the entering variable in terms of the basis.

Now we need a representation of the new entering variable in terms of the current basis. Let A_{j_e} be the column in A_N associated with the entering variable and solve the linear system

$$A_B y = A_{j_e}.$$

2.2.4 Step 4. Perform the ratio test.

We then drive a basic variable to zero by computing the maximum Θ such that

$$x_B - \Theta y \geq 0,$$

where $\Theta \in \Re$. Let

$$\Theta_i = \begin{cases} +\infty & \text{if } y_i \leq 0 \\ \frac{x_{B_i}}{y_i} & \text{if } y_i > 0 \end{cases}$$

for $i = 1, \ldots, m$, and $\Theta = \min_i\{\Theta_i\}$. If $\Theta = +\infty$, then stop, as the LP is unbounded.

2.2.5 Step 5. Update the basis.

The final step is to update the new basis. Set

$$x_B \leftarrow x_B - \Theta y.$$

Let $i_l \in \{1, \ldots, m\}$ be such that $\Theta_{i_l} = \Theta$, and let $A_{j_l} = A_{B_{i_l}}$. Then set

$$A_{B_{i_l}} \leftarrow A_{j_e}$$

$$x_{B_{i_l}} \leftarrow \Theta$$

$$A_N \leftarrow A_N + A_{j_l} - A_{j_e}.$$

2.3 Summary of work

We can summarize the work involved in a single iteration of the revised simplex method as follows:

- Solve an $m \times m$ linear system

$$\boxed{\pi^T A_B = c_B^T} \quad . \tag{2.2}$$

- Compute the reduced costs ($n - m$ inner products)

$$\boxed{d_N = c_N^T - \pi^T A_N} \quad . \tag{2.3}$$

- Solve an $m \times m$ linear system

$$\boxed{A_B y = A_{j_e}} \quad . \tag{2.4}$$

- Perform the ratio test

$$\boxed{\max\{\Theta : x_B - \Theta y \geq 0\}} \quad . \tag{2.5}$$

When $n \gg m$, the calculation of the reduced costs in equation (2.3) is the dominant cost of the computation.

2.4 The pricing pass

The calculation of the reduced costs in *Step 2* of the revised simplex algorithm is referred to as "pricing out" the nonbasic columns. The goal of calculating the reduced costs d_j is to find a nonbasic column that will decrease the value of the objective function if brought into the basis. Since any column associated with a negative reduced cost will meet this goal, there are a variety of alternatives for how this choice can be made. We discuss a few of the possibilities below.

The "traditional," or *steepest-descent*, method for choosing the best entering column is to select the column with the most negative reduced cost, $d_{j_e} = \min_{j \in N}\{d_j \mid d_j < 0\}$. While straightforward to implement, this choice usually does not lead to the best performance in practice.

Another alternative for choosing an entering column is referred to as *steepest-edge*. Geometrically, steepest-edge may be viewed as measuring the change in the objective function per unit that x moves along the edges of the polytope of feasible solutions joining the current iterate to the next iterate. Steepest-edge is more expensive per iteration than steepest-descent due to an extra solve and pricing pass required to update the *steepest-edge norms* (which in turn requires an additional division for each nonbasic variable).

Kuhn and Quandt [11] showed steepest-edge to take fewer iterations than steepest-descent, but it was largely ignored due to the increased work per iteration. Harris [10] implemented a steepest-edge type algorithm, called Devex, which showed good performance in both iteration count and time spent per iteration. The first practical exact steepest-edge algorithm was given by Goldfarb and Reid [9]. They presented formulas for the update of the square of the normalized edge vectors, greatly reducing the time spent per iteration. Using the update formulas and the new technology that allows larger problems to be solved, combined with faster methods for calculating the reduced costs [4], the computational advantages of steepest-edge became evident. Results from tests performed by Goldfarb and Forrest [8] comparing steepest-edge

to other pricing techniques show the advantages of steepest-edge in terms of both iteration count and total time required to reach an optimal solution.

For crew scheduling, steepest-edge pricing is better than standard steepest-descent pricing for both primal and dual formulations of the problem [2]. For this reason, we chose steepest-edge pricing as the pricing technique to be used by each processor on its subset of nonbasic columns.

Due to the flexibility of our parallel implementation of the simplex method, we were able to test a variant of *greatest decrease* as a selection criterion for the choice of an entering column. A "true" implementation of greatest decrease in the simplex algorithm would require that for all of the possible entering columns (all columns with a negative reduced cost) the change in the objective function value Δz be calculated. All of the objective function changes then would be compared and the column associated with the greatest decrease in the objective function would be chosen to enter the basis. However, for our implementation of greatest decrease we do not look at the objective function change for all of the columns with a negative reduced cost. Instead we look only at the objective function change for one column from each processor. Every processor calculates the steepest-edge norms locally for its own subset of nonbasic columns. Based on its subset of steepest-edge norms, each processor chooses a local column to enter the basis. Each processor calculates the corresponding objective function change for its particular choice of an entering column. A single global choice is made for an entering column by comparing the objective function change among the candidates from each of the processors. The candidate associated with the greatest decrease in the objective function is chosen to enter the basis. Since we do not look at the change in the objective function for every possible entering column, we refer to this as a variant of the greatest decrease selection criterion for the simplex algorithm.

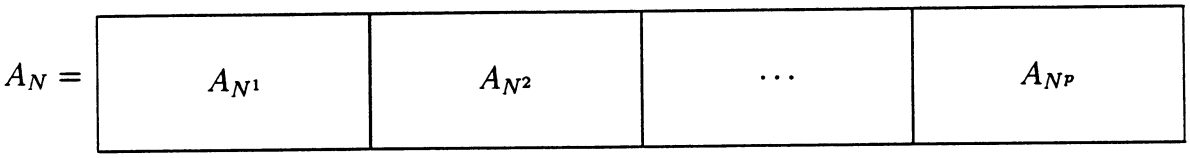

Figure 2.1 The distribution of the nonbasic columns

2.5 A description of a parallel revised simplex algorithm

Since the computation of the reduced costs for crew scheduling problems is the dominant cost of the primal revised simplex method when implemented on a sequential machine, it is our choice to parallelize this part of the computation. By dividing the set of nonbasic columns among the available processors, we will show how it is possible to exploit both parallelism and distributed memory.

2.5.1 The distribution of the data

When $n \gg m$, most of the data is contained in the nonbasic columns A_N of the constraint matrix A=[A_B A_N]. By partitioning the nonbasic columns into subsets as shown in Figure 2.1, we can distribute the columns of A_N among p processors. We are then able to "price out" the subsets of columns independently, allowing the pricing step to be carried out simultaneously. By allowing each processor to have access to the current basis and its own subset of nonbasic columns, as shown in Figure 2.2, most of the work for a single iteration of the simplex method can be performed independently on each processor. We will see that this simple strategy for distributing both the data and the computation can lead to significant reductions in the total time spent per iteration.

2.5.2 Our programming paradigm

Our implementation employs the Single Program Multiple Data (SPMD) programming paradigm. On distributed-memory machines each processor has its own local

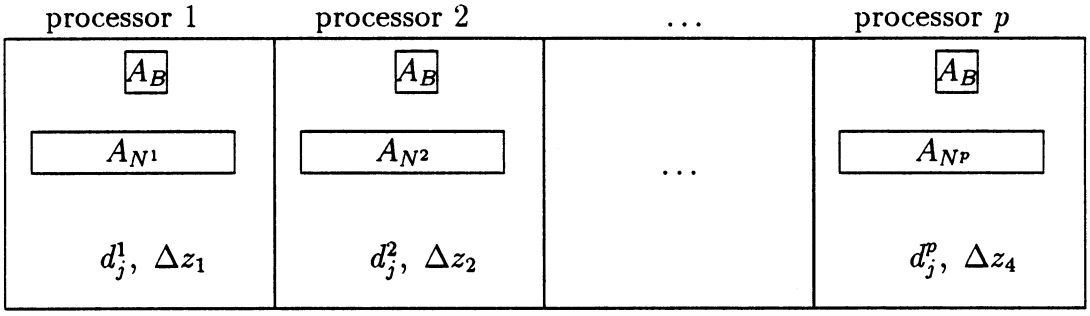

Figure 2.2 The data/memory layout for a parallel revised simplex algorithm

memory. We load the same program on each processor, hence the name single program. However, we divide the nonbasic columns into subsets so that each processor gets a subset of the nonbasic columns, hence multiple data.

During each iteration of the simplex method it is necessary to exchange information concerning the global choice of an entering column. The message is comprised of three pieces: a header (of fixed length) containing information needed for the update, the entering column, and the representation of the entering column in terms of the current basis. Thus, the message is of $O(m)$, and this length is independent of the number of processors being used. (In fact, since it is known that the columns of A have no more than eighteen nonzeros, the length of the message is considerably less than m.)

Since the nonbasic columns comprise the majority of the columns, and pricing these columns accounts for up to 99% of the work, this decomposition most efficiently distributes both the data and the work load across the processors in a way that minimizes data transfer.

2.6 A parallel revised simplex algorithm

Given the linear program in (2.1), p processors, an initial basis A_B on each processor, and the memory layout given in Figure 2.2, we will use N^i to represent the indices of the subset A_{N^i} of the columns of A_N on processor i. Thus, $N = \bigcup_{i=1}^{p} N^i$ and $|N^i| = O((n-m)/p)$. We can then restate the revised simplex algorithm.

On each processor i,

while $d_j \not\geq 0 \ \forall \ j \in N$ **do:**

Step 1: Solve $\pi^T A_B = c_B^T$. /* compute the shadow prices

Step 2: Compute $d_{N^i}^T = c_{N^i}^T - \pi^T A_{N^i}$. /* compute the reduced costs

if $d_j \geq 0 \ \forall \ j \in N^i$, **then** goto step 5.

else choose $j_e^i \in N^i$ such that $d_{j_e}^i < 0$.

Step 3: Solve $A_B y = A_{j_e^i}$. /* compute y in terms of the basis

Step 4: Compute Θ^i. /* compute the step length

Step 5: Communicate among all processors to find the "best" column.

if $d_j \geq 0 \ \forall \ j \in \{j_e^i, i = 1, \ldots, p : \ d_{j_e}^i \geq 0\} \subset N$ **then** we are optimal

else winning processor passes "best" column to losing processors.

Step 6: Update the basis.

Go to *Step 1.*

The actual implementation of this algorithm will be discussed in the next chapter.

2.6.1 The "best" entering column

There exists a multitude of techniques for pricing out the nonbasic columns. Not only can different selection criteria be used for the choice of an entering column, but techniques have also been developed that vary the subset of nonbasic columns used during the pricing pass.

In the sifting [2] implementation of Sprint [7] discussed in Section 1.2.1, a subset of nonbasic columns is selected. This subset of columns, along with the basis, is used

to formulate a problem that is solved to optimality. Then this subset is modified by adding or deleting columns and again the problem defined by this new subset of columns is solved to optimality. This process is repeated until no new columns can be added to the subset of nonbasic columns that would further improve the objective function value. This technique of using the same subset of columns over many iterations of the simplex algorithm is referred to as *multiple pricing.*

A similar pricing technique is called *partial pricing.* In this technique a different subset of nonbasic columns is priced out each iteration. The nonbasic columns comprising this subset are redefined at each iteration.

Both partial pricing and multiple pricing are in contrast to *full pricing*, where all of the nonbasic columns are priced out during the pricing pass. When $n \gg m$ the pricing pass can be the most expensive part of the simplex algorithm. Partial pricing is an attempt to speed-up this part of the algorithm. Tests comparing partial pricing[§] to full pricing, using steepest-descent, show partial pricing to be the more desirable of the two methods in iteration count as well as total time spent [5].

Examining all nonbasic columns makes sense when using steepest-edge. The introduction of an update formula for the steepest-edge norms [9] made steepest-edge a practical selection criterion. However, this update formula is effective only when using either multiple or full pricing. Since the nonbasic columns comprising a working subset are redefined at each iteration when partial pricing is used, the steepest-edge norms would have to be recomputed, rather than updated, at each iteration. It is believed [5] that for most problems full pricing would be better than multiple pricing when using steepest-edge. Since full pricing is readily accomplished in our distributed-memory implementation, this is the strategy we have employed.

By using the Intel Delta Touchstone and the distribution of the nonbasic columns illustrated in Figure 2.1, we are able to perform full pricing on problems with millions of columns. Thus, we can perform full pricing not only on problems that previously

[§]The partial pricing used in the tests is actually a combination of partial pricing and multiple pricing.

could not even have been loaded into memory, but also on problems that might have been solved using only multiple pricing due to the time required to perform full pricing. We hope that by looking at all of the nonbasic columns we can reduce the number of iterations by selecting an entering column with the best overall steepest-edge norm.

There are many criteria for selecting the column to enter the basis. However, there are no theorems that give any useful information about the number of iterations that can be expected from any of the practical pricing strategies. It is therefore important to test different pricing methods on a smaller version of a problem to get an understanding of the behavior of the simplex method. We have chosen to look at the actual change in the objective function value because intuition has long held this to be the choice that should lead to the fewest number of iterations. Given p processors and our programming paradigm, with no appreciable overhead we can compute the actual change in the objective function value for up to p possible choices of incoming variables.

Chapter 3

Implementation Details

The implementation details are many and varied and have proven to be more involved than originally expected. We will discuss only the salient features.

3.1 The LP solver

We chose CPLEX as our linear program solver. This choice was motivated by our need for both a state of the art LP solver and a stable software package. It was first necessary to modify CPLEX to run on a single processor of either the iPSC/860 Hypercube* or the Intel Touchstone Delta. Since CPLEX already runs on various platforms this was accomplished with a few straightforward modifications.

CPLEX is a complex and robust LP software package that is widely used in industry. In our efforts to modify CPLEX to run in parallel we did not want to introduce modifications that would interfere with the features that make CPLEX robust. In order to maintain these features, and to introduce as few errors as possible, our intent was to keep all modifications to CPLEX simple and introduce as few changes to CPLEX as possible. For the initial setup of the linear program we are able to use CPLEX via its callable library. However, the optimization phase required modifications to portions of the CPLEX code.

---

*the iPSC/860 Hypercube is a trademark of Intel.

3.1.1 The initial basis

The SPMD algorithm we employ requires that all of the processors start with and maintain the same basis. For this reason we load the identity matrix in the first 837 columns of each processor. In this way we establish the same initial basis on each processor before we start the optimization phase of the simplex algorithm. As a consequence, we cannot allow CPLEX to choose its own initial basis even though this has been shown to reduce the number of iterations required to complete Phase I (finding an initial feasible solution) of the simplex algorithm [3].

We force the identity to be the initial basis on all of the processors to minimize the inter-processor communication. This approach is necessary as we implemented no exchange of information among the processors during the building of the initial basis in CPLEX; otherwise, each processor would build its own initial basis from its unique set of nonbasic columns. We could possibly improve this part of our implementation by determining how to find an initial basis while preserving the need for an identical basis on all of the processors. The danger is that while the number of iterations may decrease, the total time spent finding an initial basis could increase as a result of increased communication.

3.1.2 Communication

To synchronize the processors, and thus preserve the same basis and iterates across all processors, we added a communication point to the linear program solver. This communication point is needed at the juncture in our algorithm where an entering/leaving pair of columns is known on each processor, but no basis update has been performed. This corresponds to Step 5 in our parallel simplex algorithm as outlined in Section 2.6. At this point we must decide which processor has the "best" choice for an entering column. We use an Intel communication library procedure `gopf()` to handle global communication. The procedure `gopf()` requires a user-defined subroutine to define the rules for the exchange of information. The reason for using `gopf()` is that

it sets up the communication channels and handles most of the details involved in routing global messages. To change the criterion for what determines the best column to enter the basis we only need to change the routine passed to `gopf()`. This is where the tremendous flexibility of our implementation lies.

We were also forced to introduce a second communication point because early refactorization may be triggered on one or more processors but not necessarily on all of the processors. This is due to the way in which CPLEX handles the allocation of space. If a processor finds it necessary to reallocate memory, CPLEX forces a new factorization. This can occur on just one processor since the problem size, due to different sparsity patterns for the nonbasic columns, varies from processor to processor. If one processor refactors and the others do not, the difference between a newly refactored basis and an updated basis is sufficient to cause a different leaving variable to be chosen. As noted earlier, we require all of the processors to have the same basis.

We used the Performance Analysis Tools[†] to profile the communication overhead of our implementation. The profiles show the overhead of the second communication procedure call (`gopf2`) to be small compared to the communication call to synchronize the optimization (`gopf1`). See Figure 3.1.

3.2 Primal vs. dual

For this particular problem it is known that a feasible basis can be found from among the first one thousand columns of the problem. However, this problem is highly degenerate in the primal form. By primal degeneracy we mean that there will be iterations of the primal simplex method that do not change the objective function value. During these iterations the same solution vector x is represented by a different set of basic variables and so there is no movement from the current vertex to another vertex. The best approach for solving the LP relaxation of airline crew scheduling

[†]PAT is a trademark of Parasoft and Parallel PAT is a trademark of Intel.

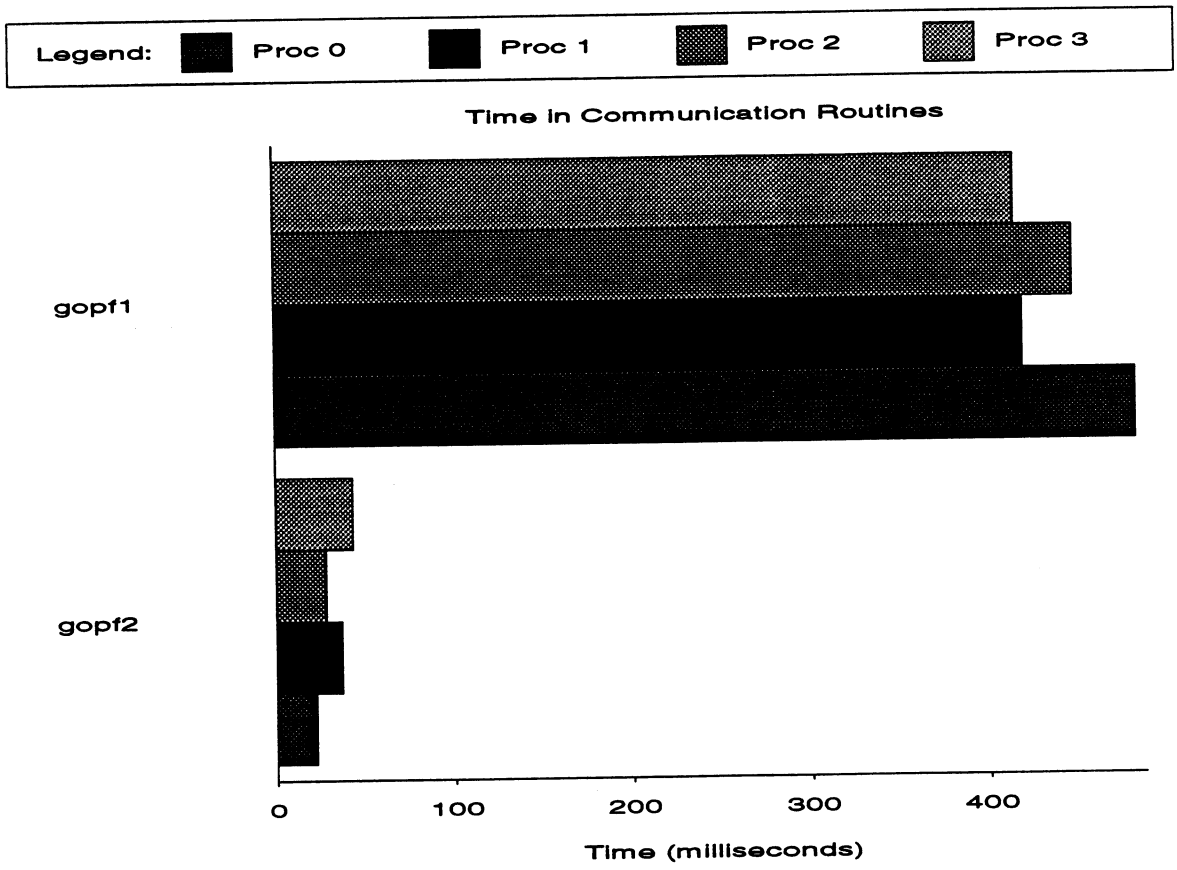

Figure 3.1 Profile of execution time for the communication routines

problems is to use the dual steepest-edge [2] simplex algorithm. However, as we mentioned in Section 1.2.3, we are interested in testing a variant of greatest decrease, and that can be tested naturally in the context of the primal. Thus, we chose to deal with this degeneracy in other ways, which we discuss below.

3.2.1 Degeneracy

In order to handle the primal degeneracy of the Challenge problem we slightly perturb the right-hand side. We want to apply as small a perturbation as possible in order to change the character of the problem as little as possible, while at the same time we

want to lessen the degeneracy. We chose a perturbation that modifies the right-hand side b as follows:

$$\begin{aligned} t &= \varepsilon * \texttt{drand}() \\ \text{if} \quad (\quad t < 0.001 \quad) \quad & t = 0.001 \\ b &= b + t \end{aligned}$$

where `drand()` is a random number generator and $\varepsilon = 0.001$.

By applying this perturbation[‡] we change the character of this problem by breaking up each degenerate vertex into a cluster of non-degenerate vertices. The result is that improvement in the actual objective function cannot occur at each iteration as long as we remain in the cluster of vertices. However, the hope is that we will leave a degenerate vertex in fewer iterations. We are able to show empirically that this is indeed the case. For a test problem of 20,000 columns executed on a Sun[§] workstation, we are able to solve the perturbed problem in 10,482 iterations and 3,165 seconds while the unperturbed problem takes 63,285 iterations and 17,849 seconds.

One disadvantage of introducing the perturbation is that we are no longer working with the original set partitioning problem. However, for testing purposes it proves to be the more rational approach, considering the time involved for solving a single instance of the problem. A second disadvantage of the perturbation became evident while testing the greatest decrease algorithm, as we will discuss further in Section 5.2.

3.3 Memory requirements

Since one of the goals of this research is to show how the entire problem can be kept in memory throughout the optimization, memory allocation is an important aspect of our work.

[‡]This is not the perturbation used in CPLEX.

[§]Sun is a trademark of SUN Microsystems.

3.3.1 Data structure

We required only one modification to the data structures of CPLEX. This is the addition of one column, of at most eighteen nonzeros, to the set of nonbasic columns on each processor. This "dummy" column, as implied by the name, acts as a place holder. At every iteration of the simplex method there is a choice as to which processor has the best entering column. Since this entering column, being nonbasic, is unique to a particular processor, it is necessary for the processor with the best entering column, which we refer to as the "winning" processor, to pass the selected column to the other processors, which we refer to as the "losing" processors. The dummy column provides the additional space needed for the losing processors to receive this column. The associated leaving column becomes irrelevant to the losing processors, once it leaves the basis, as it only needs to become a nonbasic column on the processor which originally owns the entering column. At this point the leaving column becomes the new dummy column on the losing processors, since the task of preserving the information contained in this column falls to the winning processor. Thus, the number of columns stored on each processor does not increase.

3.3.2 Compression

In the sequential version of CPLEX, the size of the problem is static once the problem is loaded. For our parallel implementation, each processor has a different set of nonbasic columns at the start of the optimization phase, and so each processor has a problem of a different size due to the differences in sparsity patterns. CPLEX employs a sparse data structure design. Hence, as the columns are loaded into memory, each column is allowed only enough space for exactly the number of nonzeros it contains. This implies that, for our parallel implementation, the start and end locations of each column in the data structure are likely to be different among the processors. Recall that the key to our parallel implementation is the exchange of columns—at every iteration the losing processors always receive a new column from the winning

processor. Since the dummy column was the leaving column in the previous iteration, there is no guarantee that the dummy column has the same number of nonzero entries allocated as the new number of nonzero entries required by the entering column. The dummy column only acts as a place holder in order to minimize the modifications to CPLEX.

CPLEX stores the constraint matrix by column. In addition, the nonbasic columns A_N are stored again by row. The latter storage is to increase the speed of the calculation of the reduced costs [5]. In order to handle the discrepancies in the size of columns among the processors we must add two types of data management to CPLEX. Even though we allocate enough total memory to handle the differences in the number of nonzeros per column, we must also be efficient with the memory we allocate. Consequently, we have added to CPLEX compression routines for both the column arrays and the row arrays. Both compression routines collect the available space that has been broken into small fragments during the change in the number of nonzero entries in the dummy column during iterations of CPLEX. This space is reallocated in subsequent iterations. Compression is done only as needed.

3.3.3 Storage

By only using as much space as required, even though the sparsity of the columns residing on a processor changes from iteration to iteration, we are able to load about 21,000 columns, along with the operating system and CPLEX, on each processor. As a consequence of work done on the Challenge problem using a Cray Y-MP [2], software exists for the removal of duplicate columns. After the duplicate columns are removed, the 12.75 million column Challenge problem reduces to 8,549,879 columns [2]. Thus, the entire Challenge problem could be stored in memory on the 512 processor Intel Touchstone Delta machine.

3.4 Selection criterion

We will use steepest-edge as our selection criterion to test our efforts at computing the reduced costs in parallel. However, we will also test an algorithmic idea that uses a different selection criterion as outlined below.

3.4.1 Across the processors

During each iteration of our parallel implementation of the simplex method, before the processors communicate with each other, each processor must calculate the reduced costs for its subset of nonbasic columns. Then each processor must select an entering column, local only to its set of nonbasic columns, as shown in *Step 2* in Section 2.6. Each processor always makes this selection based on the steepest-edge selection criterion. After each processor has made its own local selection for an entering column, the processors must communicate among themselves in order to reach a global consensus as to the best overall column to enter the basis. This corresponds to *Step 5* in Section 2.6.

By varying the global selection criterion in *Step 5* of our parallel revised simplex algorithm, we can test two algorithmic ideas. First, we will test "pure" steepest-edge, where we apply steepest-edge in both *Step 2* and *Step 5*. This allows us to test the effect of distributing the work load across the processors without introducing any significant algorithmic changes. Second, we will test a variant of the greatest decrease criterion by modifying *Step 5* to select the column associated with the largest decrease in the objective function value as the best entering column. Notice that when we use greatest decrease on only one processor, we are reduced to using the pure steepest-edge criterion since there is no communication and hence no global decision to be made.

We refer to the latter implementation as a variant of greatest decrease for two reasons. First, we do not use greatest decrease as the local selection criterion on each processor. This would probably be too expensive. Second, for our tests we use

a combination of both steepest-edge and greatest decrease for the global decision in *Step 5* which we discuss in Section 3.2.1.

3.5 Architecture

The initial development of the parallel implementation for this work was on an Intel iPSC/860 with 32 processors and eight megabytes of memory per processor. The use of this machine was provided by the Center for Research on Parallel Computation[¶] (CRPC) with support from the Keck Foundation. Further development was done on an Intel iPSC/860 with 64 processors and 16 megabytes of memory per processor. The final testing was performed on an Intel Touchstone Delta machine with 512 processors and 16 megabytes of memory per processor. The latter two machines are operated by California Institute of Technology on behalf of the Concurrent Supercomputing Consortium with access provided by the CRPC.

3.6 Test problems

3.6.1 Structured test problems

During our modifications to CPLEX on the Intel machines we performed some preliminary tests on subsets of columns from the 12.75 million column Challenge problem to gain some indication of the performance of our implementation. These structured subproblems were formed from the first 20,000 columns in the Challenge problem. Our preliminary tests made us aware of some facts about floating point division on the Intel iPSC/860. We observed that even though the number of iterations to solve these subproblems is low, the amount of wall-clock time spent is relatively large compared to the sequential version of CPLEX run on a Sun4 workstation. Profiling on the Intel iPSC/860 shows that anywhere from 20% to 35% of the total execution time

[¶]Under NSF Cooperative Agreement Nos. CCR-9120008 and CDA-8619893.

% time	seconds	cumsecs	msec/call	function
34.4	430.56	430.56	—	`.ieee_fp_div`
23.3	291.73	722.29	82.972	`_rpriceall`
19.6	246.19	968.48	91.725	`_test_s`
11.3	141.47	1109.95	76.346	`_supd2`
2.1	26.61	1136.56	32.020	`_supd0`
1.6	19.77	1156.33	7.363	`_findmin`
1.4	17.60	1173.93	21.150	`_djnormset`
1.0	13.10	1187.03	—	`_fabs`
0.9	11.37	1198.40	—	`_ftrsolve5`
0.7	8.50	1206.90	180.90	`_refactor`

Table 3.1 Profile of CPLEX on one processor of the iPSC/860

is spent performing floating point division. In Table 3.1 we have listed the routines that are the top ten contenders in percentage of execution time used. The floating point division is clearly the largest consumer of time. The next largest consumer of time is the ratio test `rpriceall`, followed by `test_s`, which computes inner products, followed by routines to update the column norms and the steepest-edge norms, `supd0` and `supd2`, respectively. Next is the routine `findmin`, which finds an eligible column to enter the basis, followed by the routine `djnormset`, which calculates the steepest-edge norms. As we noted in Section 2.4, the computation of the steepest-edge norms requires an additional division for every nonbasic variable. We assume the time required for the division in the routine `djnormset()` has been broken out into the first entry of the table. After that we find the Intel C library routine `fabs`. The last two routines, `ftrsolve5` and `refactor`, are a forward solve routine and the routine to refactor the basis, respectively. Notice that the last two routines each account for less than 1% of the total execution time. (As we mentioned earlier, the forward and backward solves do not account for much of the work when $n \gg m$.)

On the Intel iPSC/860, IEEE compliant floating point division is done in software. We profiled loops performing addition, subtraction, multiplication, and either division

Floating Point Library Operation	% Time Spent Performing Operation			
	Intel iPSC/860			Sun4
	w/IEEE		w/o IEEE	w/IEEE
	divide	reciprocate	divide	divide
__ieee_fp_div	71.1	72.4	0.0	0.0
.add	6.8	6.6	15.2	16.3
.sub	6.8	6.6	15.2	16.3
.mlt	6.7	6.4	15.2	17.6
.div	6.5	6.0	49.6	42.7

Table 3.2 Percentage of time spent in floating point loops on the Intel and Sun4 machines

or reciprocation, on both a single processor of the iPSC/860 and the Sun4 workstation. The results of the profiles show that both floating point division and reciprocation require more time on the iPSC/860 than on the Sun4. Other floating point operations are comparable on the two machines. If we disable IEEE compliance on the iPSC/860, floating point operations on the two machines are comparable. See Table 3.2.

However, we cannot afford to disable IEEE compliance when solving subsets of the Challenge problem because they are too sensitive to numerical changes. We found that when we disabled the IEEE compliance, the numerical changes were sufficient to produce a different set of iterates. The number of iterations required to reach an optimal solution increased significantly. Although the time spent per iteration decreased, due to the decrease in the amount of time spent in floating point division, the total execution time increased due to the increase in the number of iterations.

3.6.2 Unstructured test problems

In light of the slow floating point division on the Intel multiprocessor machines, we concluded that it would be impractical to attempt tests involving the entire Challenge problem.

Instead of using the entire 12.75 million columns of the Challenge problem, we believe that by properly choosing subproblems from the large problem, we can still represent the behavior of the perturbed Challenge problem with subsets of columns. Also, as mentioned in Section 1.1, it is often the case that smaller subproblems are solved in the process of solving the large problem due to the large size of the airline crew scheduling problems.

We randomly selected 20,000 column subsets from the Challenge problem. We have chosen to use 20,000 columns for several reasons. First, we can fit 20,000 columns on one processor, which allows us to compare our parallel implementation results using one or more processors of the Intel distributed-memory machines (as long as we have 16 megabytes of memory per processor). Second, a subproblem of 20,000 columns can be solved in a reasonable amount of time so that testing is practical. Third, we believe we can still test the advantages of carrying out the calculation of the reduced costs in parallel and test a variant of the greatest decrease selection criterion.

We chose to select the columns randomly for the test subproblems in an effort to better characterize the full Challenge problem. As mentioned in Section 1.1, programs are used to generate the columns. By looking at the data file for the Challenge problem, patterns can be seen in the columns that are generated. Thus, we decided to create subproblems that sample from the entire spectrum of columns of the Challenge problem. To ensure a feasible basis for each problem, the first 1000 columns of each test problem were drawn from the first 1000 columns of the Challenge problem. The remaining 19,000 columns of each subproblem were sampled randomly without replacement from the remaining columns of the Challenge problem. We used a random number generator supplied by the IMSL[‖] statistics library. We used the routine `rnsri()`, which generates a pseudorandom sample without replacement. We allowed a random seed to be obtained from the system clock and we used the default multiplier of 16807 for the generator.

[‖]IMSL is a trademark of IMSL, Inc.

Using these criteria we created 100 subproblems of 20,000 columns each. It is on 10 of these unstructured subproblems that we performed tests of our parallel implementation of the simplex method.

Chapter 4

Performance

Throughout our research we have been interested in accomplishing the following three goals:

1. Exploit distributed memory by loading the entire Challenge problem.
2. Exploit parallelism by distributing the work load.
3. Test a variant of the greatest decrease algorithm.

We will discuss what we have accomplished with respect to each of these three goals.

4.1 Distributing the data

Our first task was to load the entire Challenge problem into memory. With the use of our compression routines and careful handling of the data structures of CPLEX we are able to load about 21,000 columns per processor, along with the operating system and CPLEX. If we do not count the initial basis we force CPLEX to use, we can load about 20,000 columns of the Challenge problem per processor. On the Intel Touchstone Delta we have 512 processors which gives us sufficient memory for a total of 10,240,000 nonbasic columns across the processors. Preprocessing the data to remove duplicate columns reduces the number of columns from 12,753,313 to 8,549,879 [2]. Thus we have demonstrated that the Challenge problem can be stored across the local memory of the Intel Touchstone Delta, eliminating the need for disk I/O during the optimization phase.

4.2 Distributing the computation

Our results for exploiting the distributed memory and parallelism are conclusive. We can show a savings in average time spent per iteration by carrying out the computation of the reduced costs in parallel and hence a savings in the total time required to solve the problems. The speed-up is evident in all of our tests carried out for the "pure" steepest-edge implementation. We demonstrate these results with both a structured subproblem, shown in Table A.1 of Appendix A, and with two unstructured subproblems, shown in Appendix B.

4.2.1 Speed-up

As can be seen in Figure 4.1, which contains a graph of the average execution time per iteration taken from results given in Table A.1 of Appendix A and Table B.1 of Appendix B, the average execution time per iteration decreases as the the number of processors is doubled, up through 32 processors. As we move to more processors, the average execution time per iteration begins to slowly increase. This is true for both the structured and unstructured subproblems. It appears that by distributing the work load so that each processor computes the reduced costs for 625 nonbasic columns, the average execution time per iteration reaches a minimum.

We had hoped to see linear improvement as we increased the number of processors. Our results do not show this. As can be seen from both Figures 4.2 and 4.3, as we go from one processor to two processors we do halve the execution time. However, the linear rate of improvement does not continue when we use more than two processors. This is because the overhead of the communication outweighs the speed-up in actual computation. Ideally, if no communication were needed, fewer columns on a processor would require fewer reduced costs to be computed, and so the time spent per iteration would continue to decrease monotonically. However, it takes a minimum amount of overhead, independent of the message length, to communicate among the processors. This overhead is due to the start-up cost of sending a message and the cost of receiving

a message. So once the cost of the communication exceeds the cost of computing the reduced costs on a processor, the gain from further distributing the work load is lost to the time spent in communication. Thus, it would be faster to compute the reduced costs on fewer processors. This would increase the work load on each processor but the communication overhead would remain the same, regardless of the distributed computing environment, since the length of the messages also would remain the same. However, in order to put more columns (more work) on a processor, more memory is required per processor.

We profiled a subproblem with 40,000 contiguous columns for 2, 32, and 64 processors in order to get a better understanding of the ratio of the computational work to inter-processor communication. By choosing 40,000 columns we can load the maximum number of columns on 2 processors, which still necessitates communication, but the minimum possible. For 32 and 64 processors, we decrease the amount of work but increase the amount of inter-processor communication. The difference can be seen in Figures 4.4, 4.5, and 4.6, where we can see the fraction of time spent in either computation, communication, or idle time. The idle time usually indicates that the processor is sitting idle at a synchronization point while waiting for the arrival of a message containing information needed to continue the computation. For the 2 processor case, shown in Figure 4.4, over 90% of the time is spent performing calculations and very little in communication. However, for the 32 processor case shown in Figure 4.5, about 40% of the time is spent in computation and for the 64 processor case shown in Figure 4.6, about 30% of the time is spent in computation. Both the 32 and 64 processor cases show that a larger percentage of total execution time, relative to the 2 processor case, is spent in communication rather than in computation.

Also notice that the percentage of idle time is much greater for the 32 and 64 processor cases. In our implementation, the idle time is directly related to the percentage of time spent communicating with respect to the percentage of time spent performing computation. Every time there is a global communication call, one of the processors

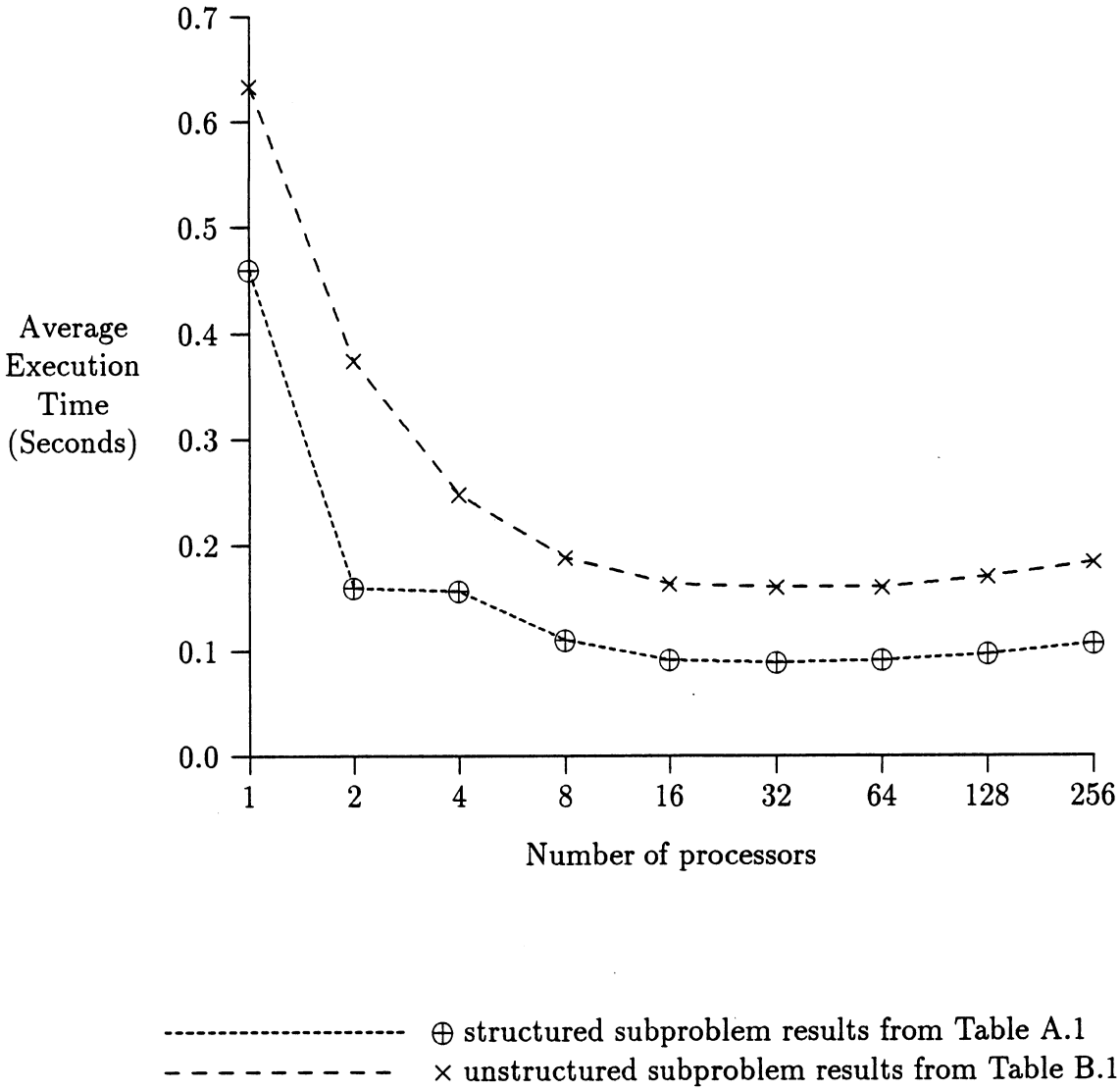

Figure 4.1 Average execution time per iteration as the number of processors is doubled for both a structured and an unstructured subproblem.

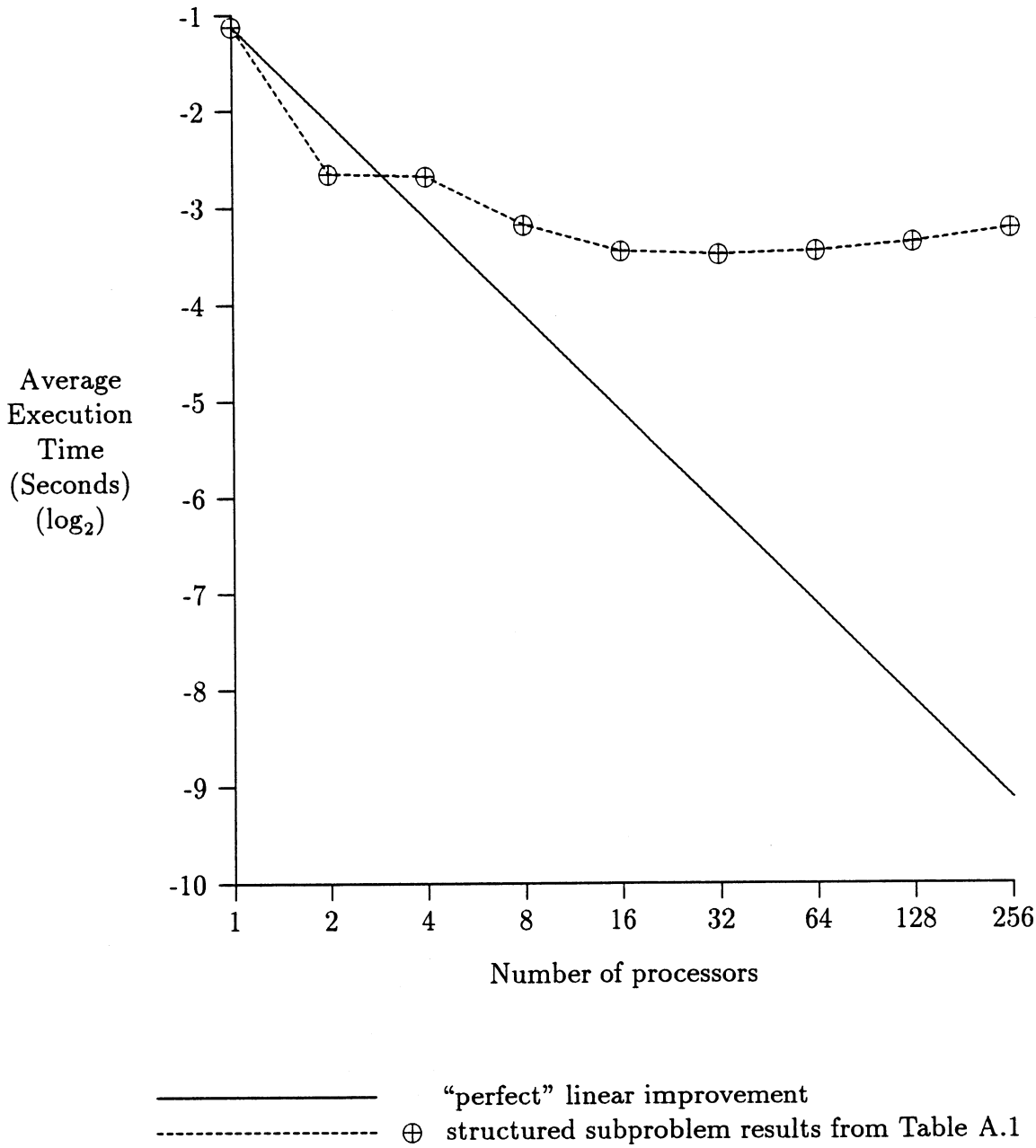

Figure 4.2 Average execution time per iteration as the number of processors is doubled for a structured subproblem

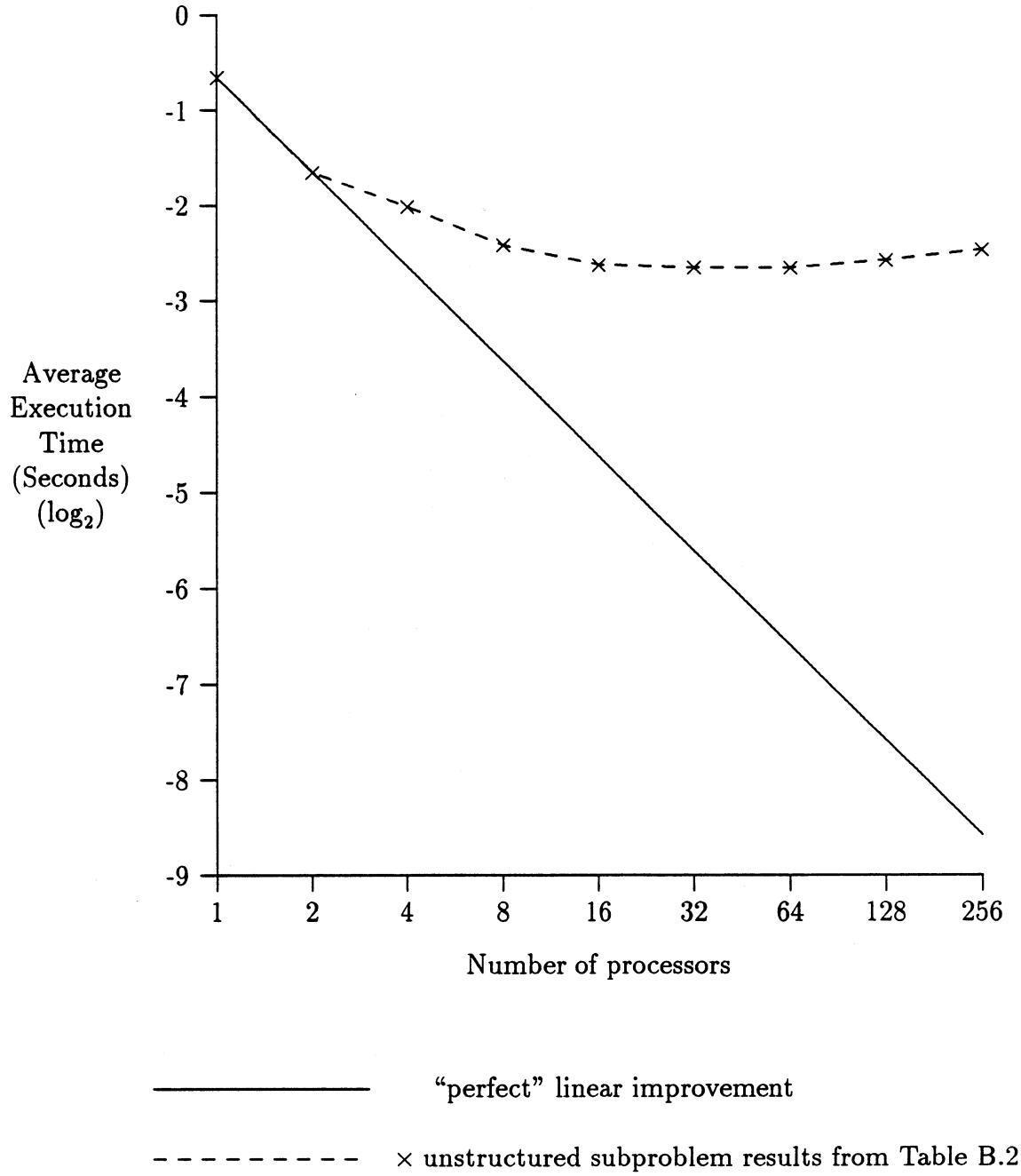

Figure 4.3 Average execution time per iteration as the number of processors is doubled for an unstructured subproblem.

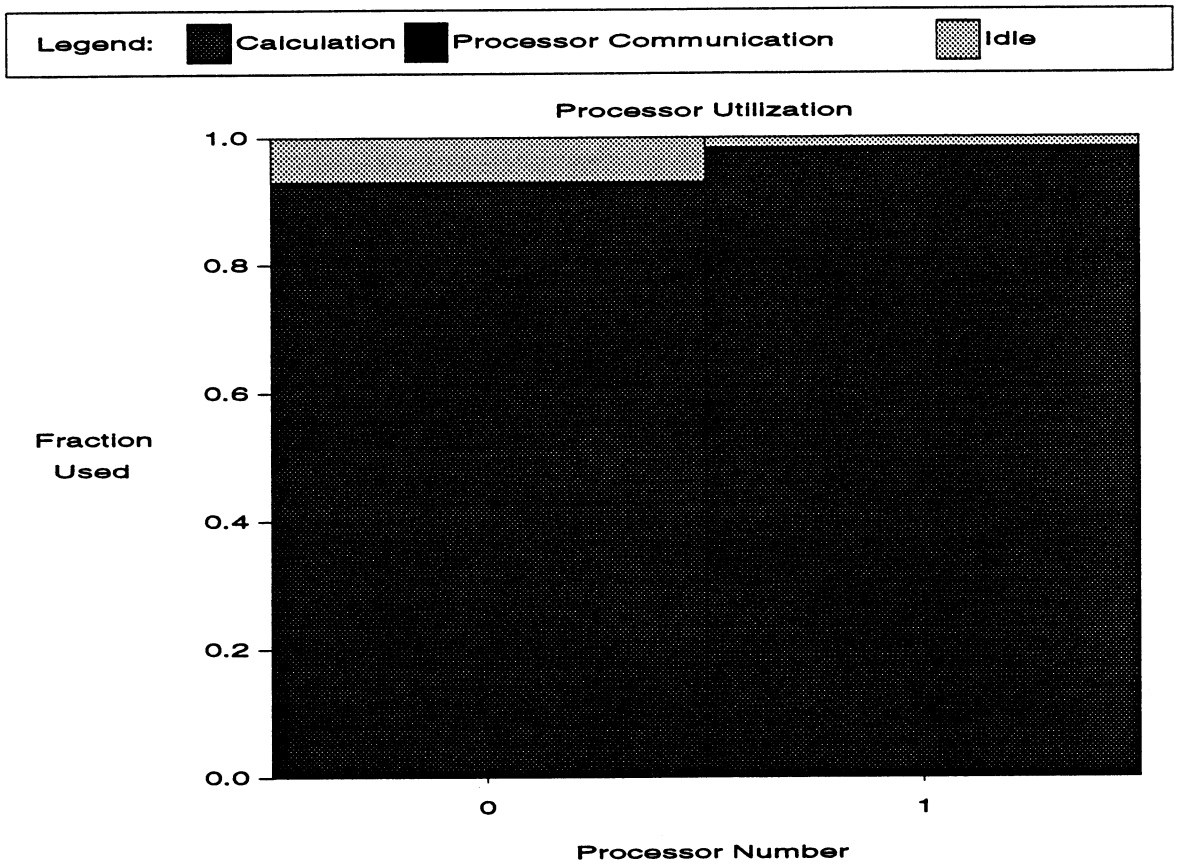

Figure 4.4 Communication profile for 40,000 columns on 2 processors

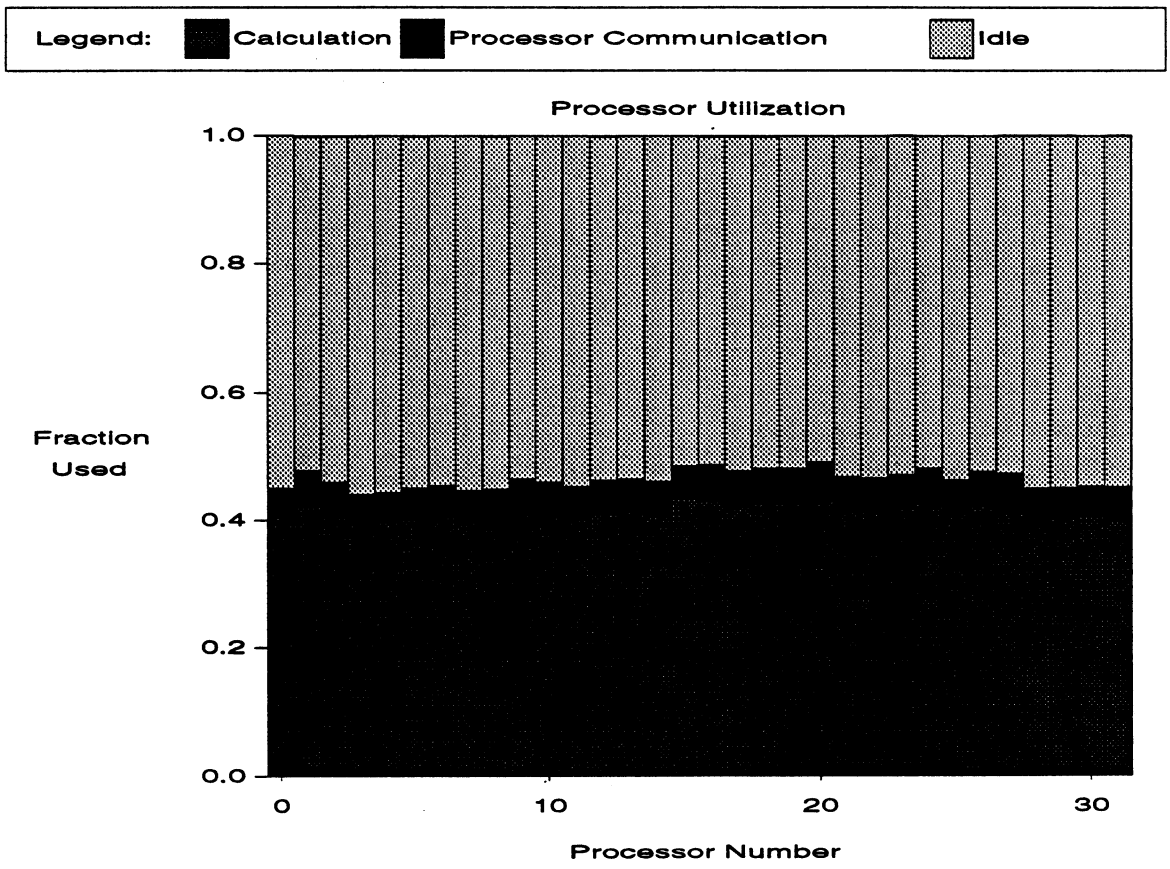

Figure 4.5 Communication profile for 40,000 columns on 32 processors

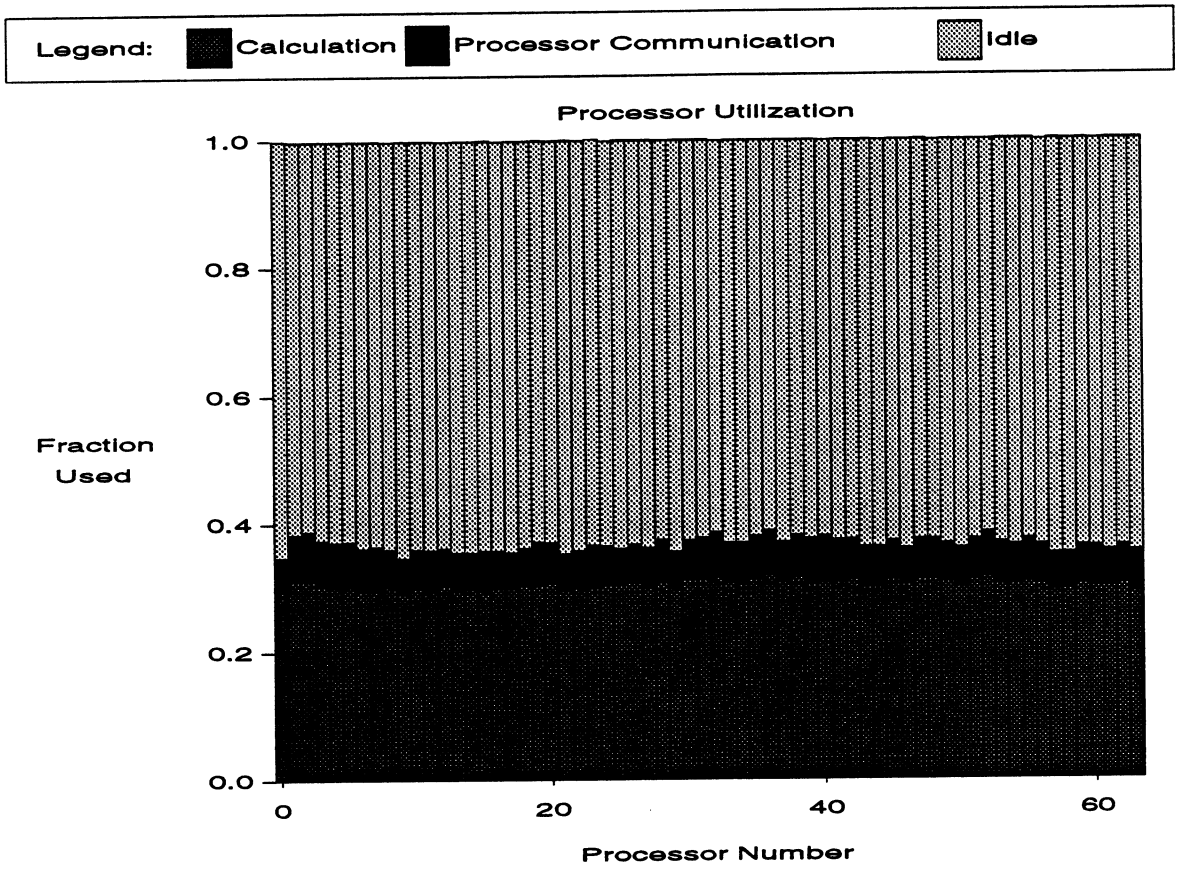

Figure 4.6 Communication profile for 40,000 columns on 64 processors

will have to wait on the other processors. This is when "losing" processors receive the entering column. These processors must do additional work to integrate the new column into the CPLEX data structures. However, the "winning" processor does not have to do any of this work and as a result gets ahead of the others and spends time waiting at the next synchronization point for the other processors to catch up. When there is only a little computation being done each iteration, the percentage of time a processor will spend idle is larger than if the amount of computation is large. Thus, we think that for our implementation a significant amount of idle time indicates that there is not enough computation to take full advantage of the processor.

Another factor that may account for the idle time observed in the profiles of the execution time on the iPSC/860 is the sparsity structure of the matrix A. The different partitions of A_N, depending on the number of processors being used, can have a different number of nonzeros. This could vary the time required for each processor to perform the linear algebra necessary to execute an iteration of the simplex method. As the number of columns per processor decreases, differences in the number of nonzero entries across the partition of the matrix can become more and more apparent. We did not perform any experiments to determine whether or not this played a role in the percentage of idle time observed as we varied the number of processors, but such behavior would be consistent with observations made by researchers working on parallel linear algebra algorithms for sparse matrices.

4.2.2 Unexpected discrepancies

There was one unexpected development. When using only steepest-edge as a selection criterion with full pricing, we would expect to take the same number of iterations to solve a problem regardless of the number of processors being used. However, looking at Table A.1 and Appendix B we see that this is not what happened. The iteration count can vary as the number of processors is varied.

This discrepancy can be explained by considering the way CPLEX handles the choice of an entering and leaving pair of columns. Once an entering column is selected, an appropriate leaving column must also be selected. If the corresponding leaving column is considered to be numerically unstable, CPLEX chooses another entering column to try to avoid the introduction of numerical instabilities or excessive infeasibilities. Our parallel implementation preserves this feature but has the effect of permitting the number of candidates to grow as the number of processors increases. This allows a different leaving column to be chosen if the number of processors is changed. Once a different leaving column is chosen, a different sequence of iterates is produced. Notice that in our test this led to only small variations in the total number of iterations required to reach the optimal solution.

4.2.3 Structured versus unstructured problems

We performed preliminary tests on some subsets of consecutive columns from the Challenge problem. Observing the inconclusive results for the greatest decrease criterion when applied to a structured subproblem (see Table A.2), motivated us to generate the unstructured subproblems from randomly selected columns of the Challenge problem. Once generated, we performed some introductory tests and found quite a change in the behavior of these subproblems. The unstructured subproblems were harder, as indicated by the increased iteration count required to reach optimality. (See Appendices A and B.) This could be further evidence that the structured subproblems do not accurately reflect the Challenge problem. Since the Challenge problem is considered a hard problem, we believe that the unstructured subproblems more satisfactorily represent its character. We conjecture that by sampling from the entire 12.75 million columns, we are able to break out of the structure created by the column generation that we encounter when we sample only consecutive columns.

Looking at Figure 4.1, a difference in the average execution time per iteration between the structured and unstructured subproblems can be observed. We can

explain this difference. For all of the results we present in Appendices A, B, and C, we tracked (on a single processor) the number of times the selected entering column had a corresponding leaving column that was considered numerically unstable by CPLEX. This is referred to as a "rejected pivot" because both the entering and leaving columns are rejected due to the numerical instability that would be introduced for a given choice of the leaving column. There is a large difference in the number of rejected pivots seen when we solve the two types of subproblems. In all of the runs in Table A.1, for the processor we tracked, there were no rejected pivots. In Table A.2, the number of rejected pivots ranges from zero to thirty. In contrast, for the unstructured subproblems in appendices B and C, the number of rejected pivots ranges from 35 to 815, considerably more than for the structured subproblems.

In CPLEX, when a pivot is rejected due to numerical difficulties, another attempt is made to find an acceptable pivot. During this effort the iteration count is not incremented, but more time is needed to complete an iteration. In our parallel implementation, a local rejected pivot on all but one of the processors requires no extra work. This is because when the global communication occurs, the processor with an acceptable pivot will win and no extra work will be required to find an acceptable pivot. Only the time required for the normal communication is needed. However, when all of the processors produce a rejected pivot, the time required for an iteration is even greater than in the sequential case. Since in the parallel case, along with the additional work of finding another pivot, there is an additional communication call among the processors every time all processors reject their pivot. The search for a stable pivot continues until at least one of the processors finds an acceptable pivot. So when there are many pivot rejections, as in the unstructured subproblems, the average time spent per iteration can be expected to increase due to both the additional computation and the extra communication overhead that rejected pivots introduce.

4.3 Greatest decrease results

Our results for using greatest decrease as the selection criterion for the choice of an entering column can be seen in Appendix C. These results show conclusively that on our test problems, the performance of the greatest decrease criterion is inferior to the performance we see using only the steepest-edge criterion. These results demonstrate that the greatest decrease rule is not appropriate for this class of problems for reasons we will discuss in the next chapter. However, our results do not rule out the possibility that the greatest decrease rule may be very effective for other linear programming problems.

Chapter 5

Conclusions

5.1 Hardware platform

We think that we have demonstrated the value of a parallel implementation for the primal simplex algorithm for problems, like the Challenge problem, with a disproportionate ratio of columns to rows.

Our experiments lead us to conclude, however, that neither the iPSC/860 nor the Touchstone Delta is the appropriate platform for our parallel implementation of CPLEX. The relatively slow speed of the IEEE floating point division is certainly one serious hindrance. The relatively small amount of local memory on each processor is also a drawback. If we were able to put more columns on each processor, we would eliminate the need for as many processors. We would thereby increase the work load on each processor. The processors would then spend a significant percentage of total execution time computing and relatively little or no time communicating or sitting idle, which would improve the performance of our implementation. We think a more suitable hardware platform would be a multiprocessor machine with much more memory per processor and fewer processors, and in particular, processors with faster IEEE floating point division.

5.2 Perturbation

As discussed in Section 3.2.1, one disadvantage of applying a small perturbation to the right-hand side vector is that we are no longer working with the original

set partitioning problem. However, there is a second drawback, which only became apparent during the testing of the greatest decrease criterion.

Recall that the point of the perturbation is to break a degenerate vertex into a cluster of non-degenerate vertices. With the introduction of these clusters of vertices, there will be some iterations of the simplex method where we simply move from one vertex in a cluster to another vertex in the same cluster. While we may see decrease in the value of the objective function, this is an artifact introduced by the perturbation and does not indicate real progress toward a solution to the underlying problem. Thus the use of the greatest decrease criterion to choose the step at these iterations seems inappropriate because there is no substantive change in the objective function value; it may simply lead to a random choice of which vertex in the cluster to take. At these iterations the change in the objective function value is very small, on the order of 0.02. We decided that a more intelligent course of action would be to use the steepest-edge criterion to choose the step at these iterations and then use greatest decrease once we move on to a "real" vertex. In order to accomplish this we looked at the objective function values from iteration to iteration and declared that any change less than 0.1 in the objective function was not a "real" decrease. At these iterations we used steepest-edge as the criterion for choosing an incoming column. Unfortunately, this did not substantially decrease the number of iterations required to reach a solution.

For all of the results shown in Table A.2 and Appendix C, we tracked the number of iterations where we used the objective function change, as opposed to steepest-edge, as the selection criterion in *Step 5* of the parallel algorithm outlined in Section 2.6. The objective function change selection criterion is used anywhere from 75% to 85% of the time. As we varied the tolerance used for making the decision between the two selection criteria, we could see that the more often we used steepest-edge, the fewer the iterations. In fact, for the tolerances we tried, the pure steepest-edge algorithm always performed better than our variant of the greatest decrease algorithm.

5.3 Poor performance of the greatest decrease algorithm

From our results we conclude that the Challenge problem is an inappropriate problem for testing the use of the greatest decrease selection criterion for the primal simplex algorithm. Due to the perturbation that we introduced, we made it very difficult to take advantage of information we have about the change in the objective function. We hypothesize that this is because we have been unable to distinguish a "real" vertex from a vertex introduced by the perturbation. The greatest decrease criterion, with a different set of test problems, may still prove to be an effective selection criterion for the primal simplex algorithm.

Appendix A

Structured Subproblem Results

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	65945.63805555557	0	2684	1234.569	.460
2	65945.63805555557	0	2684	427.511	.159
4	65945.63805555557	0	2684	419.269	.156
8	65945.63805555557	0	2684	295.342	.110
16	65945.63805555557	0	2684	242.925	.091
32	65945.63805555560	0	2716	241.204	.089
64	65945.63805555560	0	2716	247.474	.091
128	65945.63805555560	0	2716	264.359	.097
256	65945.63805555560	0	2716	291.747	.107

Table A.1 "Pure" steepest-edge on the first consecutive 20,000 columns (structured subproblem).

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	65945.63805555557	0	2684	1234.569	.460
2	65945.63805555555	1849	2409	623.363	.259
4	65945.63805555565	2022	2681	437.949	.163
8	65945.63805555561	1885	2533	278.318	.110
16	65945.63805555568	2151	2614	235.673	.090
32	65945.63805555551	2111	2637	230.773	.088
64	65945.63805555557	2587	3336	300.002	.090
128	65945.63805555548	2754	3440	331.790	.097
256	65945.63805555542	2795	3486	369.258	.106

Table A.2 Greatest decrease on the first consecutive 20,000 columns (structured subproblem).

Appendix B

Steepest-edge Results

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	65717.19239809163	0	10290	6519.987	.634
2	65717.19239809162	0	9632	3609.874	.375
4	65717.19239809162	0	10305	2552.488	.248
8	65717.19239809152	0	10172	1914.676	.188
16	65717.19239809168	0	10309	1678.065	.163
32	65717.19239809168	0	9841	1563.562	.160
64	65717.19239809162	0	9969	1594.455	.160
128	65717.19239809153	0	10299	1747.440	.170
256	65717.19239809171	0	10262	1884.913	.184

Table B.1 Steepest-edge: unstructured subproblem 001

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66094.91566015418	0	10427	6591.083	.632
2	66094.91566015419	0	10484	3926.223	.374
4	66094.91566015415	0	10626	2615.001	.246
8	66094.91566015419	0	10266	1924.380	.187
16	66094.91566015416	0	10339	1724.924	.167
32	66094.91566015416	0	10484	1662.591	.159
64	66094.91566015412	0	10379	1643.302	.158
128	66094.91566015413	0	10560	1811.040	.172
256	66094.91566015421	0	10512	1920.054	.183

Table B.2 Steepest-edge: unstructured subproblem 010

Appendix C

Greatest Decrease Results

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	65717.19239809163	0	10290	6519.987	.634
2	65717.19239809149	8434	11157	4187.060	.375
4	65717.19239809152	10209	12715	3178.437	.250
8	65717.19239809160	11289	13840	2588.218	.187
16	65717.19239809155	11470	14367	2369.465	.165
32	65717.19239809150	12969	15902	2543.461	.160
64	65717.19239809156	14021	17047	3067.408	.180
128	65717.19239809152	14741	18068	3067.415	.170
256	65717.19239809156	16950	20693	3817.308	.184

Table C.1 Greatest decrease: unstructured subproblem 001

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66077.71548219235	0	10525	6617.980	.629
2	66077.71548219242	8724	11227	4189.122	.373
4	66077.71548219242	9749	12351	3067.330	.248
8	66077.71548219243	11037	13818	2575.164	.186
16	66077.71548219242	11164	13908	2386.451	.172
32	66077.71548219239	11955	14925	2379.531	.159
64	66077.71548219238	14415	17682	3274.842	.185
128	66077.71548219239	16531	19637	3223.585	.164
256	66077.71548219238	17511	21521	3843.697	.179

Table C.2 Greatest decrease: unstructured subproblem 002

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66101.61413186065	0	9878	6189.371	.627
2	66101.61413186068	8690	11285	4220.139	.374
4	66101.61413186071	9592	12348	3046.552	.247
8	66101.61413186062	10990	13695	2586.761	.189
16	66101.61413186067	10877	13789	2247.069	.163
32	66101.61413186061	12535	15737	2518.323	.160
64	66101.61413186076	13798	16903	2949.535	.174
128	66101.61413186067	14662	18059	2954.884	.164
256	66101.61413186062	15934	19193	3391.625	.178

Table C.3 Greatest decrease: unstructured subproblem 003

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66112.85548290343	0	10468	6610.214	.631
2	66112.85548290337	8509	10977	4124.044	.376
4	66112.85548290338	9354	12118	3036.102	.251
8	66112.85548290335	10429	13457	2592.094	.193
16	66112.85548290340	11241	14066	2398.138	.170
32	66112.85548290329	12784	15802	2435.731	.154
64	66112.85548290335	13947	17206	2731.309	.159
128	66112.85548290334	14197	17368	2826.900	.163
256	66112.85548290331	15389	18752	3399.980	.181

Table C.4 Greatest decrease: unstructured subproblem 004

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66376.85627889386	0	10591	6746.184	.637
2	66376.85627889370	9293	12023	4447.371	.370
4	66376.85627889373	9390	12022	2963.093	.246
8	66376.85627889380	10819	13557	2580.055	.190
16	66376.85627889371	11036	13898	2221.268	.160
32	66376.85627889374	12664	15712	2436.612	.155
64	66376.85627889374	13100	16152	2553.328	.158
128	66376.85627889379	16022	19377	3151.887	.163
256	66376.85627889370	16200	19483	3392.364	.174

Table C.5 Greatest decrease: unstructured subproblem 005

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66005.84856133825	0	10367	6550.703	.632
2	66005.84856133827	8272	10770	4039.500	.375
4	66005.84856133837	9861	12575	3062.024	.244
8	66005.84856133827	10585	13491	2561.471	.190
16	66005.84856133832	11767	14562	2456.353	.169
32	66005.84856133828	13220	16604	2631.601	.158
64	66005.84856133829	14431	17874	2827.507	.158
128	66005.84856133819	15987	19469	3208.854	.165
256	66005.84856133824	15539	19138	3345.102	.175

Table C.6 Greatest decrease: unstructured subproblem 006

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66136.12833348897	0	10287	6484.084	.630
2	66136.12833348889	8685	10961	4098.645	.374
4	66136.12833348893	9639	12256	3075.952	.251
8	66136.12833348893	10498	13814	2615.174	.189
16	66136.12833348894	12334	15427	2533.612	.164
32	66136.12833348894	12369	15404	2436.466	.158
64	66136.12833348893	14135	17234	2761.006	.160
128	66136.12833348894	16047	19210	3160.061	.165
256	66136.12833348893	16200	20087	3564.081	.177

Table C.7 Greatest decrease: unstructured subproblem 007

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	65563.59864988379	0	10453	6631.488	.634
2	65563.59864988376	8337	10779	3985.153	.370
4	65563.59864988389	9818	12498	3124.463	.250
8	65563.59864988377	9987	12760	2422.021	.189
16	65563.59864988383	11995	15206	2546.963	.167
32	65563.59864988382	12915	16259	2560.018	.157
64	65563.59864988389	13087	16491	2592.382	.157
128	65563.59864988379	15358	18891	3087.990	.163
256	65563.59864988383	16401	19836	3515.729	.177

Table C.8 Greatest decrease: unstructured subproblem 008

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66085.06057166310	0	10574	6698.875	.634
2	66085.06057166311	8682	10967	4085.566	.373
4	66085.06057166313	9764	12373	3074.874	.249
8	66085.06057166305	10692	13397	2497.217	.186
16	66085.06057166315	12693	15812	2623.420	.166
32	66085.06057166310	12234	15112	2295.331	.152
64	66085.06057166308	14096	17753	2905.369	.164
128	66085.06057166313	14461	17692	2928.163	.166
256	66085.06057166305	16124	19715	3432.533	.174

Table C.9 Greatest decrease: unstructured subproblem 009

no. processors	objective fcn. value	no. g.d. its.	no. its.	time (secs.)	secs./it.
1	66094.91566015418	0	10427	6591.083	.632
2	66094.91566015410	8661	11604	4372.319	.377
4	66094.91566015419	9238	11964	3011.774	.252
8	66094.91566015416	10746	13633	2576.826	.189
16	66094.91566015415	11663	14799	2467.613	.167
32	66094.91566015413	13194	16417	2517.444	.153
64	66094.91566015416	14202	17585	2809.573	.160
128	66094.91566015410	15387	18717	3112.354	.166
256	66094.91566015410	16832	20462	3707.164	.181

Table C.10 Greatest decrease: unstructured subproblem 010

Bibliography

[1] J. Barutt and T. Hull. Airline crew scheduling: Supercomputers and algorithms. *SIAM News*, 23, 1990.

[2] R.E. Bixby, J.W.Gregory, I.J.Lustig, R.E.Marsten, and D.F.Shanno. Very large-scale linear programming: A case study in combining interior point and simplex methods. Technical Report 91-11, Rice University, Department of Mathematical Sciences, Houston, Texas, 1991.

[3] Robert E. Bixby. Implementing the simplex method: The initial basis. Technical Report 90-32, Rice University, Department of Mathematical Sciences, Houston, Texas, 1990.

[4] Robert E. Bixby, 1993. From private communications.

[5] Robert E. Bixby. Progress in linear programming. Technical Report 93-40, Rice University, Department of Computational and Applied Mathematics, Houston, Texas, 1993.

[6] V. Chvátal. *Linear Programming.* W. H. Freeman and Company, 1983.

[7] J.J. Forrest. Mathematical programming with a library of optimization routines. Presentation. *ORSA/TIMS Joint National Meeting New York*, October 1989.

[8] John J. Forrest and Donald Goldfarb. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992.

[9] D. Goldfarb and J.K. Reid. A practicable steepest-edge simplex algorithm. *Mathematical Programming*, 12:361–371, 1977.

[10] P.M.J. Harris. Pivot selection methods of the Devex LP code. *Mathematical Programming*, 5:1–28, 1973.

[11] H.W. Kuhn and R.E. Quandt. An experimental study in the simplex method. *Mathematical Programming*, XV, 1963.