To the Graduate Council :

I am submitting herewith a dissertation written by R. P. Sundarraj entitled "Distributed Algorithms for Linear Programming." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Management Science.

James Ho

J. K. Ho, Major Professor

We have read this dissertation
and recommend its acceptance :

Lori Kaplan

Kenneth C. Gilbert

[illegible]

Accepted for the council :

[illegible]

Vice Provost
and Dean of the Graduate School

Distributed Algorithms

For

Linear Programming

A Dissertation Presented

for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

R. P. Sundarraj

August 1990

To my parents

ACKNOWLEDGEMENTS

I would first like to express my deep gratitude to Professor James K. Ho for his valuable guidance, encouragement and support not only during the course of this work but also during my entire graduate career. I would also like to thank each one of the members of my doctoral committee, Kenneth C. Gilbert, Lori A. Kaplan and Gordon R. Sherman for their advice and useful feedback in the finalization of this dissertation.

ABSTRACT

In recent years there has been an increasing use of parallel computers to solve problems in many scientific and engineering disciplines. The impact of parallel computers on large-scale computing is beginning to affect the area of mathematical programming. Our primary goal in this dissertation is to adapt the revised simplex method for solving linear programs on a class of parallel computers, called distributed computers.

The work consist of two parts. First, we study the serial revised simplex algorithm and provide mathematical characterizations of how time is spent in the algorithm and its modules. We refer to such characterizations as timing models. In the second part of the work, we design revised-simplex-based algorithms for tasks such as reinversion, dual vector computation, pricing, pivot row selection and solution updating. Using the timing models, the speedup provided by these algorithms are inferred. We select 35 medium-sized problems from NETLIB [Gay (1985)] to test and validate our timing models and algorithms. Over the test problems, the speedups provided by our algorithms on the iPSC/2 (an example of a distributed computer) range from 5% to 41% and is 31% on the average.

TABLE OF CONTENTS

CHAPTER		PAGE
1.	INTRODUCTION AND OVERVIEW	1
2.	PARALLEL COMPUTERS	7
3.	IMPLEMENTATION OF THE REVISED SIMPLEX METHOD	29
4.	TIMING MODELS	40
5.	DISTRIBUTED ALGORITHMS	80
6.	FUTURE WORK	131
BIBLIOGRAPHY		135
APPENDIX		141
VITA		145

LIST OF TABLES

TABLE		PAGE
4.1	Results with timing model for RSM	52
4.2	Effect of economic inversion frequency on run time	54
4.3	Effect of nonoptimality of economic inversion frequency	57
4.4	Densities of PFI etas	64
4.5	Results with timing model for BTRAN	64
4.6	Results with timing model for PRICE	69
4.7	Results with timing model for CHUZR	75
4.8	Weighted errors Vs values of parameters	76
4.9	Results with timing model for UPDATE	79
5.1	Communication times for the iPSC/2	82
5.2	Storage requirements for the DINV algorithm	95
5.3	Communication mechanisms for the DINV algorithm	95
5.4	Results with the DINV algorithm	109
5.5	Results with the DPRI algorithm	119
5.6	Inference of run time of the DPRI algorithm	120
5.7	Results with the DCHUZR and DUPDT algorithms	126
A.1	Listing of test problems	143

LIST OF FIGURES

FIGURE		PAGE
2.1	Flynn's taxonomy of computers	9
2.2	Schematic and operation of an array computer	12
2.3	Operation of vector computers	14
2.4	Schematics of shared and distributed memory computers	16
2.5	Hypercubes of dimension $n \leq 3$	18
3.1	Permuted basis during reinversion	34
4.1	Illustration of the models' errors	44
4.2	Definition of INVERT cycle	48
4.3	Growth pattern of the etas in the revised simplex method	48
4.4	Growth pattern of iteration time	50
4.5	BTRAN at consecutive iterations	59
5.1	Graph of communication times for the iPSC/2	82
5.2	Message sending and receiving process	84
5.3	Flowchart of the DINV algorithm	87
5.4	Eta file profile at consecutive slave dispatches	88
5.5	Effect of basis permutation on the DINV algorithm	90
5.6	DINV process on distributed memory computers	101
5.7	DPRI algorithm using two processors	116
5.8	Flowchart of the DBTRAN algorithm	128

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1 Objective

In the 1960's, advances in IC (Integrated Circuit) silicon chip technology made it possible to build faster and smaller computers. Realizing that the speed of the IC chips will eventually reach their performance limits, computer designers turned to parallelism to improve the computational performance of computers. This led to computers with multiple processors. In contrast to the conventional (serial) computers, these new type of computers could perform many computations in parallel and were referred to as *parallel computers*. Parallel computation is typically achieved in two ways. One can have all the processors executing the same instruction synchronously but on multiple data or one can have the processors operate asynchronously on their own instruction and data. The latter kind of parallel computers are often referred to as *distributed computers*.

The objective of this dissertation is to address the question : "How can the revised simplex method be adapted to a distributed computing environment ?"

1.2 Introduction

In the 1970's, parallel computers were available only at exclusive research centers. The ILLIAC IV, one of the first parallel computers, was put into operation at NASA's Ames

Laboratory in 1972 and the CDC STAR-100 became operational in Lawrence Livermore Laboratory in 1974. In recent years, however, there has been increasing use of parallel computers to solve large-scale computing problems. Problems in many fields such as, weather forecasting, electronic circuit design, image processing and molecular chemistry, considered beyond the scope of serial computers are now amenable due to parallel computation (see Hwang and Briggs (1984) or Ortega and Voigt (1985)). In the 1980's, parallel computers have reached the stage where they are considered as basic research tools in many scientific and engineering disciplines.

The use of parallel computation will typically speedup the solution of a problem over conventional (serial) solution procedures. However, there are several issues that are unique to a parallel programming environment. Techniques to deal with these issues are treated by many authors (e.g. Flynn (1987) and Schendel (1984)). Among these issues are the following :

(a) The problem solving task has to be distributed among the processes of the program. Suppose in a parallel program the work distribution is not uniform. Then processes with less work will be idle more than the other processors and such a program is not efficient. Therefore, the load among the processors should be as evenly balanced as possible.

(b) In a parallel program, the activities on the different processors must be coordinated and this requires the communication of information among the processors. The programmer must try to minimize the amount of interprocessor communication or must at least make the communication as efficient as possible.

(c) From a programming standpoint, serial computers essentially differ from one another in their processing speed. On the other hand, parallel computing architectures are often specialized to perform some type of computation efficiently; for example, the

CRAY has specialized units for performing vector products efficiently. Therefore, to adapt an algorithm to a given type of parallel computer, one must exploit some feature of the computer's hardware. This entails knowledge of the specific computing architecture.

1.3 Overview

The impact of parallel computers on large-scale scientific computing is beginning to affect the field of operations research. In the area of linear programming, parallelization efforts are progressing along two major directions. First, researchers have parallelized decomposition methods (e.g. Dantzig-Wolfe decomposition) which lend themselves naturally to parallel computation. Second, the revised simplex method for solving linear programs has been adapted to a class of parallel computers, called vector computers, that have specialized units for executing vector products efficiently. Our primary goal in this dissertation is to design revised simplex-based algorithms to solve LP's on distributed computers.

Algorithms designed for distributed computers typically consist of multiple processes executing concurrently the subroutines (or copies of a subroutine) of a program . Such algorithms are called *distributed algorithms*. Suppose the solution of a problem requires the execution of two tasks A and B which take the amount of time t_A and t_B respectively. Assume that B can start only after A is completed. Then, if A and B are partitioned among n processors, the time for such a distributed algorithm is :

$$(t_A+t_B) / n \; + \text{Interprocessor communication time}$$

However, if A and B are independent tasks, then they can be executed concurrently on two processors and the time for this algorithm is :

$$\text{Max}\{t_A, t_B\} + \text{Interprocessor communication time}$$

In the first case, the same task is distributed among the processors and hence such algorithms are called homogeneous distributed algorithms. Algorithms of the second type are called heterogeneous distributed algorithms.

In this work we design distributed algorithms of both types for the various tasks of the revised simplex method. These tasks are basis reinversion (INVERT), pricing (PRICE), pivot row selection (CHUZR), solution updating (UPDATE) and dual vector computation (BTRAN). The algorithms are named distributed INVERT, distributed PRICE and so on. Distributed INVERT (DINV) and distributed BTRAN are examples of heterogenous distributed algorithms while the others belong to the category of homogeneous distributed algorithms.

We wanted to predict the speedup provided by our algorithms. For this purpose, we study the revised simplex algorithm and derive mathematical characterizations of how processing time is spent by the algorithm and its tasks. Such a characterization shall be referred to as a *timing model*. The timing model for the revised simplex method is based on our observation that the iteration times between reinversions can be described by a linear model. Based on this fact, the saving in the wall-clock time to solve a problem using the DINV algorithm is predicted. Speed-up predictions are also made for the other distributed algorithms. Inferred savings on two of our algorithms

are significant and hence they are implemented on the iPSC/2, an example of a distributed computer.

For the empirical testing of our timing models and distributed algorithms, we selected 35 test problems from NETLIB [Gay (1985)]. The problem sizes ranged from 205 rows to 1481 rows. The tests serve to validate our timing models and computational results with our algorithms are encouraging. The (approximate) savings on the run-time to solve our problems range from 5% to 41% of the serial time and the average saving is 31%.

1.4 Organization of the Dissertation

In chapter 2, we provide an introductory survey of the area of parallel computing. First, the taxonomy of computers due to Flynn (1966) is reviewed. Discussions in chapter 5 require the details of programming the iPSC/2 and Sequent Balance, the two computers considered in this work. These details are discussed in chapter 2. In chapter 3, we review specifications for a robust implementation of the revised simplex method.

In chapter 4, we develop timing models for the revised simplex method and its modules. Empirical results validating the models are reported. In chapter 5, we describe distributed algorithms developed for the tasks of the revised simplex method. The description of each of the algorithms is organized as follows. First, we provide a generic statement of the algorithm without discussing machine dependent aspects of its actual implementation. A design of its implementations on the iPSC/2 and the Sequent

Balance is then described. Next, we derive the method used to infer the possible speedup of this design. Finally, computational results are reported.

In chapter 6, we identify topics for further research. A brief description of the test problems used in this research is provided in the Appendix.

CHAPTER 2

PARALLEL COMPUTERS

In the 1950's and 1960's, increases in the computational speed of computers was achieved by means of faster hardware devices. The first generation of computers were made of vacuum tubes. Since then, significant speed increases were achieved through the use of transistors and IC (Integrated Circuit) silicon chips.

Realizing that the hardware parts of computers will eventually reach their performance limits, some designers turned to radically new ways to improve the computational performance of computers. While the computers of the 50's and 60's were uniprocessor machines (serial computers), the new computers had more than one processing element and were called *parallel computers.*

In this chapter, we provide an introductory survey to the area of parallel computing. In §2.1, we describe a commonly-used taxonomy of computers. This taxonomy yields insight into the success or failure of an algorithm on a given type of computer. In §2.2, the measures used to evaluate the performance of parallel programs are discussed. In §2.3, we give an introductory treatment of a class of parallel computers. The discussions in Chapter 5 require the details of programming the iPSC/2 and the Sequent Balance, the two computers considered in this work. The necessary background information regarding these two computers is provided in §2.4.

2.1 Flynn's Taxonomy of Computers

Parallel computers come in many diverse architectures (as of 1983 there were at least 55 designs according to Schwartz (1983)). This diversity adds a new element of complexity to solving large-scale problems. Before the advent of parallel computers, the question confronting researchers was : "Which algorithm is most efficient for solving a given problem?" Today, the question is : "Not only which algorithm, but also which parallel system is most suitable for implementing the chosen algorithm?" Although there are no exact answers to these questions, Flynn's taxonomy of computer systems sheds some light into the adaptability of an algorithm to a given computer.

A serial computer performs one instruction at a time and that instruction involves one datum. Flynn (1966) observed that the methods for achieving parallel operation depended on the manner of replicating the instruction stream and data stream (see figure 2.1). This gives rise to four classes of computers. The single-instruction single-data (SISD) computer is the serial computer mentioned above. An example is the VAX 11/785. By replicating only the data stream we create the single-instruction multiple-data (SIMD) computer, considered by some authors (Hockney and Jesshope (1981)) as the only truly *parallel* computer. These computers are suitable for applications requiring vector and matrix computations. Examples of SIMD computers are : ILLIAC IV [Barnes (1968)], Connection Machine [Hillis (1985)], ICL-DAP [Hockney (1980)], Cray X-MP and IBM-3090. Figure 2.1 depicts a multiple-instruction single-data (MISD) computer. The fourth class, called MIMD computers, combines parallelism in both the instruction and data streams and is often referred to as a *distributed* computer. A parallel program designed for a MIMD (or

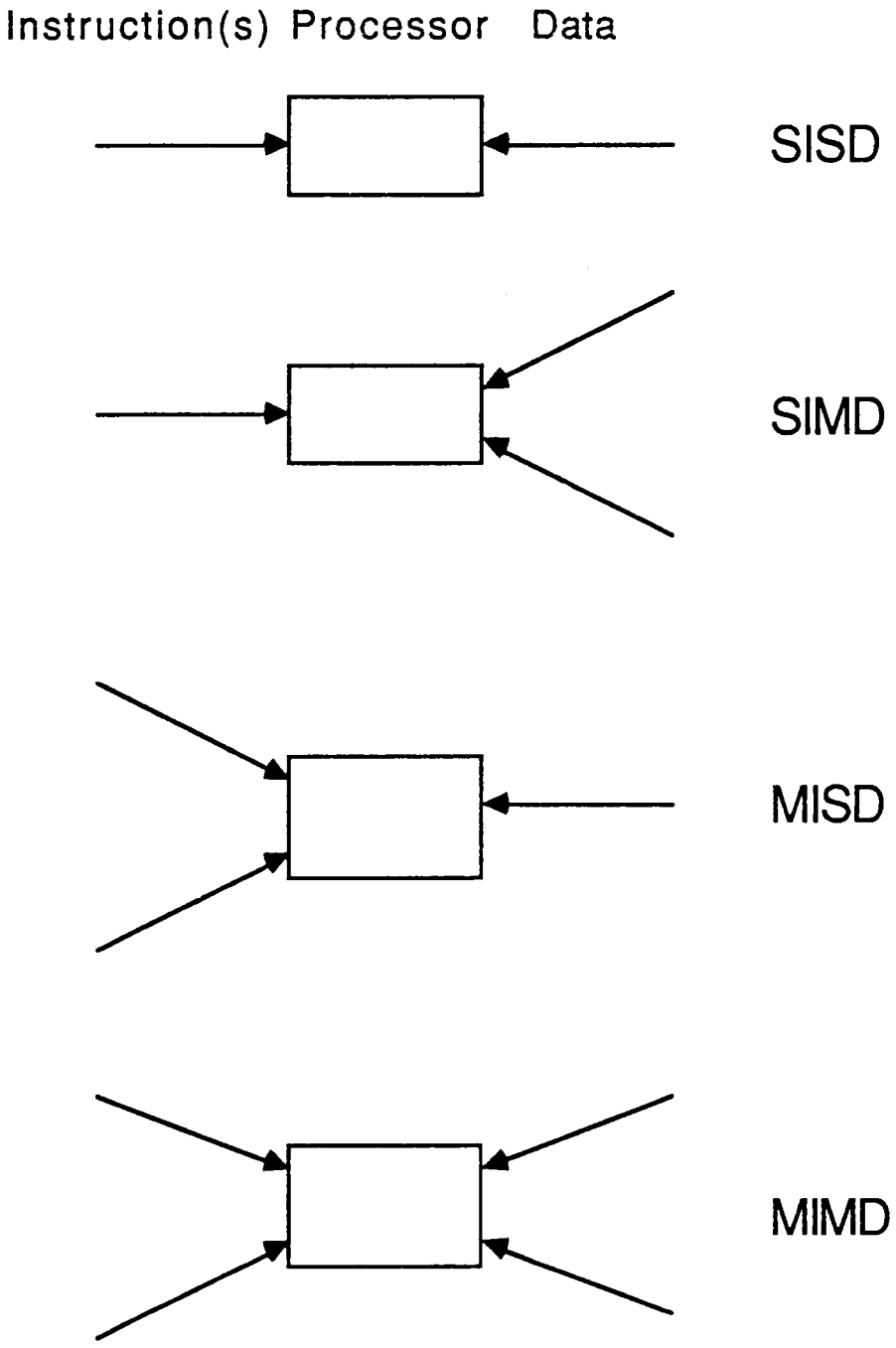


Figure 2.1 : Flynn's taxonomy of computers

distributed) computer consists of multiple processes executing the subroutines (or copies of a subroutine) of the program concurrently. That is, the parallelism exploited is at the subroutine-level and such a parallelism is often referred to as *coarse-grained parallelism*. Examples of MIMD computers are : Intel iPSC/2, Sequent Balance and CRYSTAL [DeWitt, Finkel and Solomon (1987)].

2.2 Performance Measures for Parallel Algorithms

The main purpose of parallel computation is to reduce the elapsed time (wall clock time) taken to solve a problem. The reduction in the elapsed time is measured by *speedup*, defined as the ratio of the solution time of the algorithm executing on a single processor (serial time) to the solution time of the same algorithm when executing on multiple processors (parallel time). Ideally, we would like the speedup of an algorithm to be linear with the number of processors employed for the algorithm. A superlinear speedup usually implies that the parallel algorithm has taken a different and a more efficient solution path than the serial algorithm. In practice, the speedup is usually sublinear due to processor idle time caused by interprocessor communication overhead, an unevenly balanced workload among the processors and the sequential nature of parts of the algorithm. Amdahl's law [Amdahl (1967)] provides a bound on the speedup gained from parallel computation. Mathematically, the law can be stated as follows. Let p be the number of processors, k be the fraction of code that is sequential and (1-k) be the fraction executing in parallel. The maximum speedup of the algorithm is then bounded by $1 / [k + (1-k)/p]$.

Knowing the speedup, it is reasonable to ask how efficiently the parallel

computing system is being utilized by the algorithm. One way to accomplish this is to use the *efficiency* measure, defined as the ratio of the speedup of the algorithm to the number of processors employed for achieving this speedup. Thus in the ideal situation of a speedup of p for p processors, the efficiency is unity. But, if k% of the program is sequential, then the speedup bound provided by Amdahl's law implies that the efficiency is bounded by 1 / [k*p + (1-k)].

2.3 Single-Instruction Multiple-Data Computers

A single-instruction multiple-data (SIMD) computer consists of multiple processors executing the same instruction but on different data. The SIMD computer is efficient for applications which require a lot of vector computations. There are two broad subclasses of SIMD computers : the array computer and the vector computer.

An array computer, as the name indicates, has an array of processors (see figure 2.2). Parallelism of vector operations is achieved when each vector element is treated by a different processor. For example, to add vectors x and y, the addition of the elements x(i) and y(i) is done by the ith processor. A vector computer has specialized units, called functional units, to perform *vector arithmetic*. The operation of a functional unit on vectors is analogous to that of an automobile plant's assembly line on a series of cars. The computation process, like the car-building process, is broken up into many stages. Just as there are many cars in the assembly line each at a different stage of completion, there are many elements of the vector operands in the functional unit each at a different stage in the computation process. For example, if the computation process of addition has been broken up into three stages, then the ith,

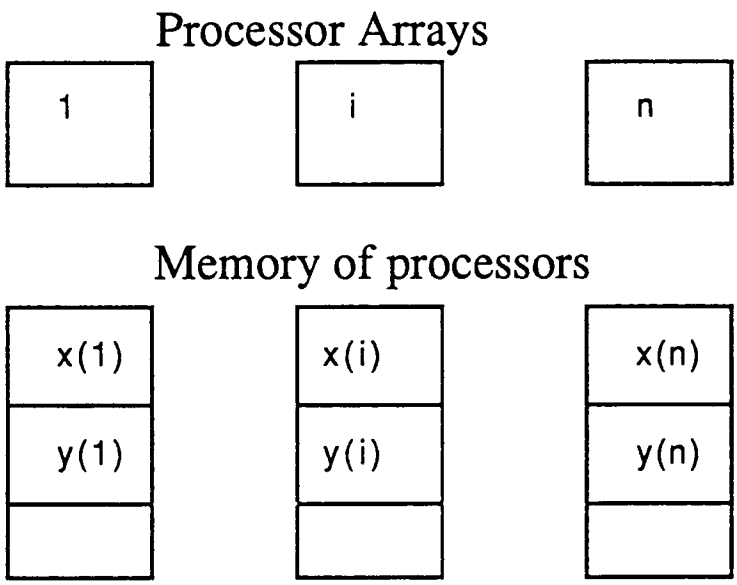


figure 2.2 : Schematic and operation of an array computer

(i-1)st and (i-2)nd elements of the vector would be in stages 3, 2 and 1 respectively.

The programming of array computers is considered in depth by Kuck (1968). Many authors have dealt with programming vector computers (often called *vectorization*). See for example Dongarra and Eisenstat (1984), Neves (1984) and Zenios and Mulvey (1986),(1987). The key *difference* between array and vector computers is illustrated by the following example. Consider adding two floating point n-dimensional vectors x and y. The task of adding any pair of elements in the above vector is typically broken up into three subtasks each of which we will assume, for simplicity, to take the same amount of time (say t) to complete. The subtasks are : (1) compare exponents of the vectors x and y; (2) shift x so that exponents are equal and (3) add the mantissas of x and y. An array computer with n processors would take 3t time units, a vector computer would take $t*n$ time units and a serial computer would take $3t*n$ time units. On an array computer the subtasks associated with the operation x(i)+y(i) are performed by the ith processor ($1 \leq i \leq n$). The ith processor is comparing the exponent of x(i) with that of y(i) during the first t time units, is shifting x(i) during the second t time units and adding the mantissas of x(i) and y(i) during the third t time units. On a vector computer, however, different elements of the vector operands are at different stages of the addition. During, say, the third period, x(3) and y(3) are in the first stage of the addition process (exponent comparison), x(2) is in the second stage (exponent shifting) and x(1) and y(1) are in the final stage (mantissa addition). Figure 2.3 illustrates this process.

Examples of single-instruction multiple-data computers are : ICL DAP [Hockney and Jesshope (1981)], ILLIAC IV [Barnes (1968)], Connection Machine [Hillis (1985)], IBM 3090, Cray 1 and Cray X-MP.

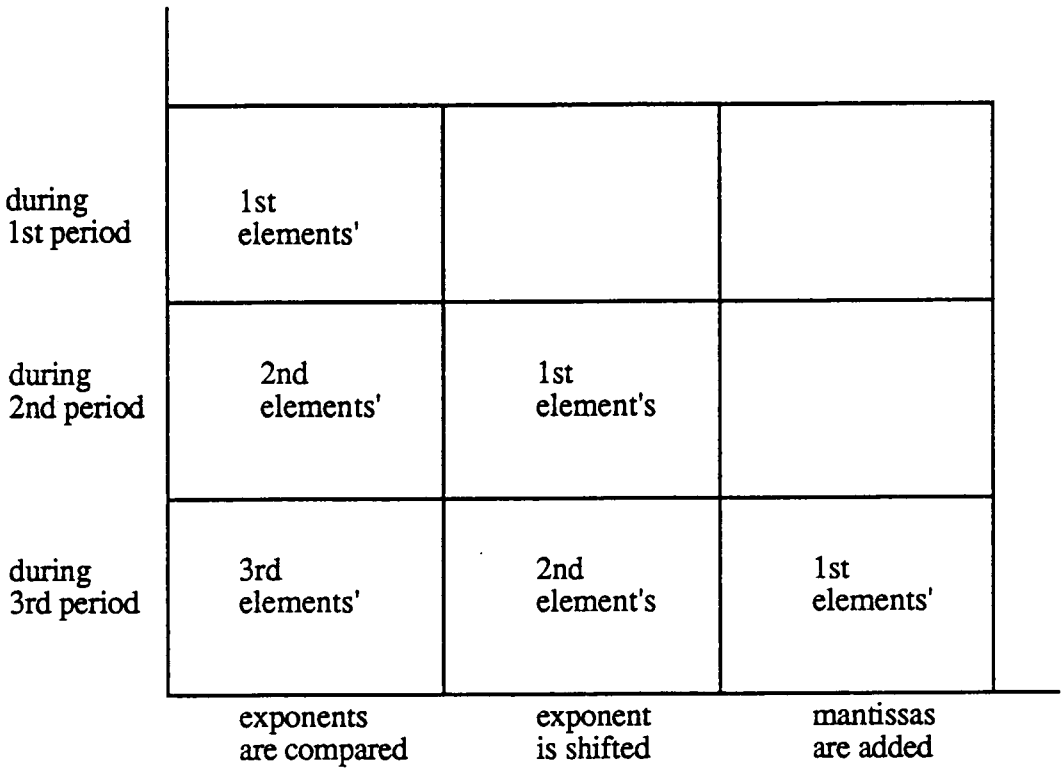
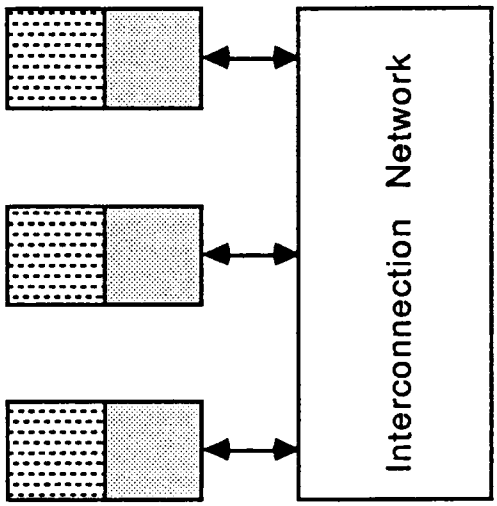


Figure 2.3 : Operation of vector computers

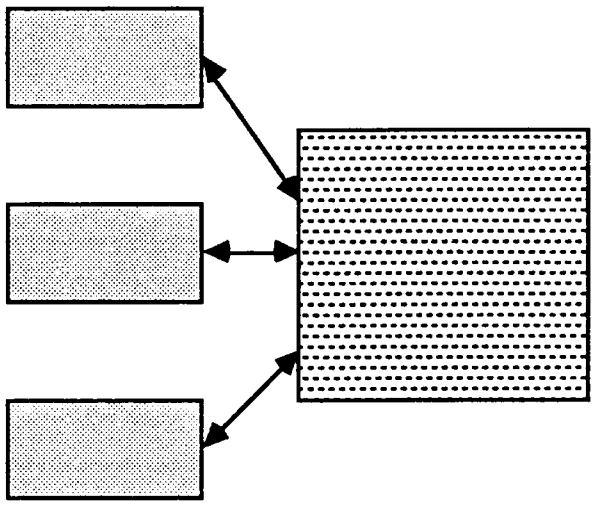
2.4 Multiple-Instruction Multiple-Data Computers

A multiple-instruction multiple-data (MIMD) computer consists of multiple processors each operating asynchronously on its own instruction and data streams. A MIMD computer is often referred to as a *multicomputer* or a *distributed* computer, since each of its processors is essentially an SISD computer. MIMD computers are generally suitable for applications that exhibit coarse-grained parallelism. It is relatively straightforward to adapt an algorithm to these computers. However, to attain high-efficiency in computation using an MIMD computer, one has to address problems of task synchronization and scheduling. There are two types of MIMD computers : the distributed memory computers (e.g. iPSC/2) and the shared memory computers (e.g. Sequent Balance).

Each processor in a distributed memory computer has its own local memory but no memory is globally accessible to all processors (see figure 2.4). Distributed memory computers are suitable for coarse-grained applications that require little communication between processors. Interprocessor communication is achieved by passing messages through a communication network. The time to communicate a floating point number from one processor to another is significantly higher than the time taken to perform a floating point computation. According to Heath et al (1987), it is about 25 times for the iPSC/1. The design of the communication network is therefore crucial to the performance of distributed memory computers. A fully connected scheme is the best as far as communication efficiency is concerned. However, such a scheme is impractical due to engineering limitations arising from the



Distributed Memory Computer



Shared Memory Computer



Figure 2.4 : Schematics of shared and distributed memory computers

very large number of connections required. An interprocessor connection scheme that is becoming popular among researchers is the hypercube scheme due to Seitz (1985). In a hypercube topology the processors are connected as they would if they lie at the corners of a multidimensional cube. The number of processors in an n-dimensional (d-n for short) hypercube is 2^n. For example, eight processors would be connected in a 3-cube and sixteen processors would be connected in a topological arrangement that models a four dimensional cube. The schematics of d-n hypercubes for $n \leq 3$ are illustrated in figure 2.5. As with any distributed memory MIMD computer, interprocessor communication on the hypercube is done by passing messages. A processor can directly communicate with its n neighbors. Messages to non-neighbors are routed through intermediate processors. The hypercube topology provides an efficient balance between the cost of communication and the benefits of direct linkage. Usually, a host computer serves as an administrative console and as a "gateway" to the hypercube for users. This is illustrated in figure 2.5 for a d-3 hypercube.

A shared memory computer has a single global memory accessible to all processors (see figure 2.4). In addition, each processor may have a small amount of local memory. These computers are effective for coarse-grained applications that require a high degree of interprocessor communication. Communication between processors is fast, essentially at hardware speed, and is done by setting variables in a memory bank that is shared by all the processors. However, to program these computers one has to be on the lookout for contentions that arise when two processors want to access the same memory location simultaneously. To handle this, the system provides the LOCK-UNLOCK construct which can be used for giving exclusive access by only one processor at a time; the LOCK prevents access to a memory unit(s) and the UNLOCK allows access to it.

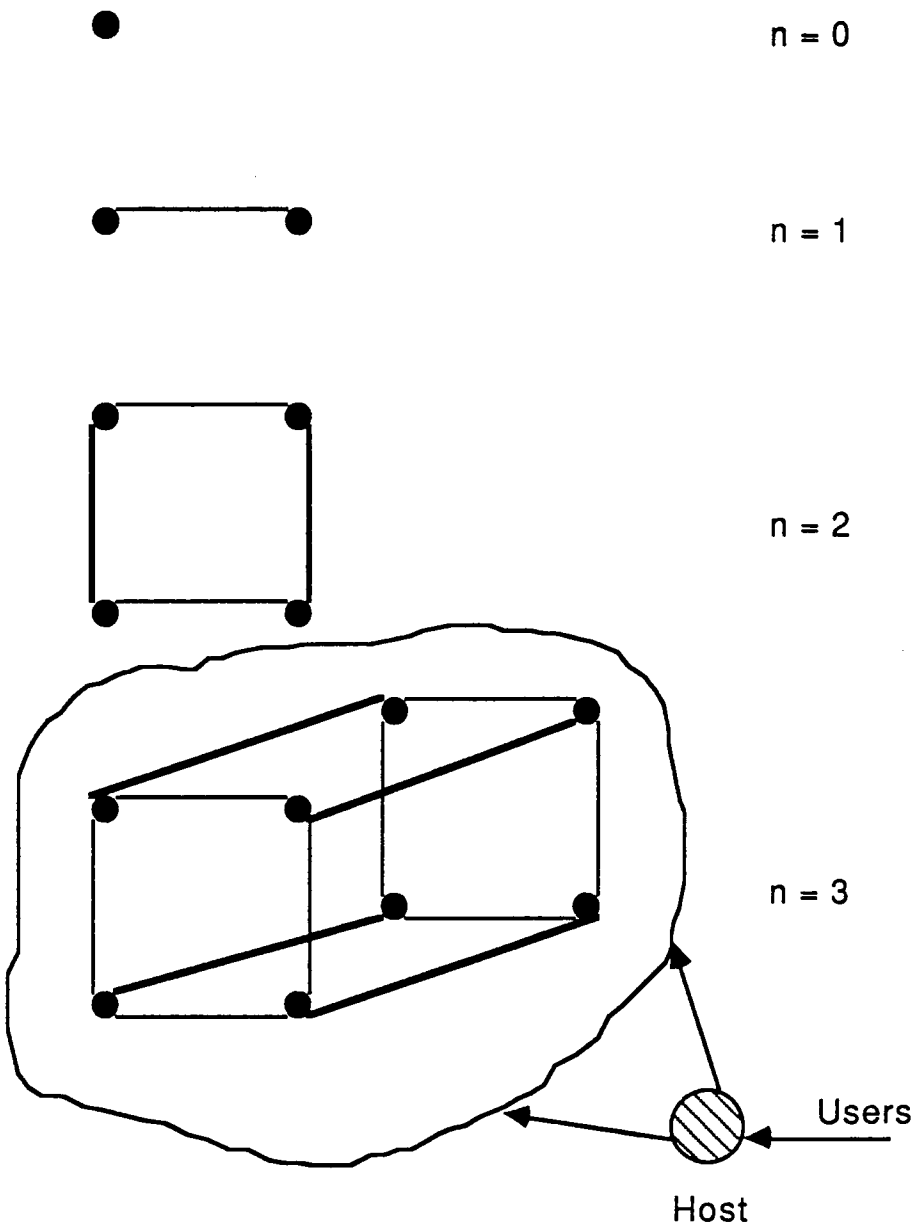


Figure 2.5 : Hypercubes of dimension $n \leq 3$

2.4.1 The iPSC/2 : A Distributed Memory Hypercube Multicomputer

Two of the distributed algorithms (cf. Chapter 5) derived in this dissertation have been implemented on the iPSC/2 multicomputer. The details concerning the architecture and programming of this computer is necessary to understand the description of those implementations. These details are discussed herein.

The iPSC/2, manufactured by Intel, is a distributed memory MIMD hypercube multiprocessor. Each processor consists of Intel's 32-bit 80386 CPU (4 MIPS) coupled with a 80387 (300 Kflops) numeric coprocessor for floating point acceleration. It has 4 MB of local memory. The hypercube (or cube) is accessed via a host (or system resource manager) which is also a 80386-based system with 8 MB memory and a 140 MB hard disk. The operating system on the host is the UNIX system V/386 (release 3.0).

The host machine allows the user to perform the following tasks.

a. To edit and link host/node programs

b. To access and release the cube (or a partition thereof)

c. To execute the host program.

d. To start or kill processes on the cube.

Operations unique to the cube are controlled either by UNIX-type commands (iPSC/2 commands) or by extensions to standard programming languages (iPSC/2 routines). The iPSC/2 commands are input from a terminal and can be used to gain access to the cube, to load, start or kill cube processes and to relinquish access to the cube. The iPSC/2 routines, on the other hand, are mainly used to manage internodal messages. Nevetheless, it should be noted that almost all of the tasks that can be performed with the iPSC/2 commands can also be accomplished from within the user programs by

iPSC/2 routines with similar names. The iPSC/2 commands and routines for FORTRAN programming are documented in Intel (1988a). A subset of the subroutines and their functions are listed below.

SETPID(PID)	Called by host to set the process ID.
KILLCUBE()	Called by host to kill processes
LOAD('exec_prgm', NID, PID)	Called by host to load node programs.
CSEND(MID, MSG, LNTH, DEST,PID)	Blocking send
CRECV(MID, MSG, LNTH)	Blocking receive
ISEND(MID, MSG, LNTH, DEST, PID)	Non-blocking send
IRECV(MID, MSG, LNTH)	Non-blocking receive
INFOCOUNT()	Length of the last message received
IPROBE(MID)	Indicates if message has been received
MYNODE()	Indicates the ID of the node
MYHOST()	Indicates the host ID
MCLOCK()	Gives elapsed time since node was initiated

Here PID is the process id, exec_prgm is the name of an executable program, NID is the node id, MID is the message type, MSG is the pointer to the message, LNTH is the message length and DEST the destination node ID. To execute a typical parallel program, the following steps are used.

a. Compile and link the host and node programs to create executable modules.

b. Obtain a partition of the cube (a subcube) of suitable size by invoking the GETCUBE command. The user has the option of providing a name to identify this partition. For example, the iPSC/2 command : getcube -c sugar -t d3 allocates to the

user an exclusive subcube named "sugar" with dimension 3 (that is 8 processors) identified by the indices 0, ... ,7.

c. Run the host program which in turn loads the node programs at runtime in response to the LOAD routine called in the host program.

d. On termination, kill all node processes and flush messages by invoking the KILLCUBE command.

e. Relinquish access to the subcube by the RELCUBE command.

2.4.1.1 iPSC Example

Consider the task of adding an array of numbers. Suppose that we use two nodes (or processors) in addition to the host. We require two programs, called *host* and *node*, running on the host and the two nodes respectively. The first part of the host program reads the numbers and distributes them between the two nodes. The program on the nodes adds its portion of the data and sends the partial sum to the host which then sums up the partial sums and writes the result. Given below are the *host* and *node* programs.

```
        host :
        REAL*8 SUM, TEMPSUM, A(50)
        INTEGER*4  SIZE
        COMMON/ADD/ SUM, TEMPSUM, A, SIZE
C
C     PREPARE DATA FOR ADDITION
```

```
      SIZE = 20
      DO 10 I = 1, 50
            A(I) = I
10    CONTINUE
C
C     LOAD NODE PROGRAM node  TO NODES 0 & 1.  PROCESS ID IS 0
      CALL LOAD('node', 0, 0)
      CALL LOAD('node', 1, 0)
C
C     SEND PARAMETERS TO THE NODES 0 AND 1.  MESSAGE TYPE IS 1
C     PID IS 1.  THE SIZE IS 8*50 FOR THE "A" PLUS 4 FOR "SIZE"
      CALL CSEND(1, A, 404, 0, 0)
      CALL CSEND(1, A, 404, 1, 0)
C
C     WAIT FOR THE NODES TO SEND THEIR RESULTS.  MESSAGES ARE
C     RECEIVED INTO TEMPSUM AND BOTH OF TYPE 2
      SUM = 0.
      CALL CRECV(2, TEMPSIZE, 8)
      SUM = SUM + TEMPSUM
      CALL CRECV(2, TEMPSIZE, 8)
      SUM = SUM + TEMPSUM
C
C     WRITE OUT "SUM" AND KILL ALL PROCESSES
      WRITE(6,*) 'THE SUM IS ', SUM
      CALL KILLCUBE(-1,0)
```

```
      STOP
      END


      node :
      REAL*8 TEMPSUM, A(50)
      INTEGER*4  SIZE
      COMMON/ADD/ TEMPSUM, A, SIZE
C
C     RECEIVE MESSAGE STARTING AT  "A(1)" TYPE IS 1 & LENGTH IS 404
      CALL CRECV(1, A, 404)
C
C     FIGURE OUT WHAT TO ADD AND DO THE ADDITION
      I1 = 1
      IF (MYNODE() .EQ. 1) I1 = SIZE + 1
      I2 = SIZE
      IF (MYNODE() .EQ. 1) I2 = 2*SIZE
      TEMPSUM = 0.
      DO 10 I = I1,I2
            TEMPSUM = TEMPSUM + A(I)
10    CONTINUE
C
C     SEND "TEMPSUM" TO HOST; MESSAGE TYPE IS 2; LENGTH IS 8; PID IS
C     0;
```

```
CALL CSEND(2, TEMPSUM, 8, MYHOST(), 0)
END
```

2.4.2 The Sequent Balance 8000 : A Shared Memory Multicomputer

In Chapter 5, the implementation of distributed algorithms on the Sequent Balance is discussed. The architectural and programming details of the Sequent Balance, required for discussions in Chapter 5, are described below.

The Sequent Balance 8000 (SB for short) is a shared memory multicomputer, i.e., a computer having multiple identical processors and a single common memory. Current models can have from 2 to 12 processors and can be configured with 4 to 28 MB of shared storage. Each CPU of SB is a 32-bit microprocessors and has 8 KB of local memory and 8 KB of cache memory. For secondary storage, the SB system offers both disks and tape drives. The system runs the DYNIX operating system, a version of UNIX 4.2bsd.

The system provides language extensions which can be coded from within the program. A subset of subroutines and their functions are given below.

M_SET_PROCS(P)	- Allocate P processors
M_GET_MYID()	- Return process id
M_FORK(SUB)	- Execute "SUB" in parallel
M_PARK_PROCS()	- Suspend child process execution
M_SYNC()	- Check in at a predefined barrier
M_LOCK()	- Lock a lock

M_UNLOCK() - Unlock a lock

M_KILL() - Kills all processors

Since all the processors are identical, a user can be logged on to any one of them. The user then edits the program and compiles it using the command (for FORTRAN) : fortran -e -F/SHRD/ prg.f -lpps, where "SHRD" is the name of a shared COMMON block and prg.f is the name of the parallel program. The pseudo-code below illustrates the coding of parallel programs on the SB system.

```
Main Program
    1. Call M_SET_PROCS(p)    /* Number of child processes */
    2. Execute sequential part
    3. Execute 'sub1' in parallel
        a. Call M_FORK('sub1')      /* Spawn p child processes */
        b. Call M_PARK_PROCS()      /* Suspend child processes */
    4. Execute the next sequential part
    5. Execute 'sub2' in parallel
        a. Call M_FORK('sub2')      /* Spawn p child processes */
        b. Call M_KILL()            /* Kill child processes */
    6. Execute sequential part, print answers and stop
Subroutine sub1
    1. Execute the parallel part
    2. Call M_SYNC() to synchronize all cooperating processes
Subroutine sub2
    1. Execute the parallel part
```

2. Call M_SYNC()

In many cases it may be necessary to ensure that a shared data structure is in use by only one process at any given time. We call such a data structure a shared *critical* section. The M_LOCK-M_UNLOCK provides exclusive access to a critical section by one process at a time. MLOCK() is called to lock a critical section and M_UNLOCK() to unlock it. The access of a critical section by a process requires the following steps. If the critical section is already in use by another process, then the M_LOCK call (step 1) puts the the first process into a wait state until the critical section is unlocked (M_UNLOCK) by the second process.

1. Call M_LOCK()
2. Use the critical section
3. Call M_UNLOCK()

We now code the example problem described in §2.4.1.1 (iPSC example).

```
      REAL*8 SUM, A(50)
      INTEGER*4  SIZE
      COMMON/ADD/ SUM, A, SIZE
C
C     PREPARE DATA FOR ADDITION
      DO 10 I = 1, 50
            A(I) = I
10    CONTINUE
```

```
      SIZE = 20
C
C     CREATE 2 CHILD PROCESSORS AND SET THEM OFF WITH TASK
      CALL M_SET_PROCS(2)
      CALL M_FORK(SUMER)
C
C     WRITE SUM OUT AND KILL ALL PROCESSOR
      WRITE(6,*) 'THE SUM IS ',SUM
      CALL M_KILL()
      STOP
      END
C
      SUBROUTINE SUMER
      REAL*8 TEMPSUM
      REAL*8 SUM, A(50)
      INTEGER*4  SIZE
      COMMON/ADD/ SUM, A, SIZE
C
C     DECIDE ON THE WHAT TO ADD AND DO THE ADDITION
      I1 = 1
      IF (M_GET_MYID() .EQ. 1) I1 = SIZE + 1
      I2 = 20
      IF (M_GET_MYID .EQ. 1) I2 = 2*SIZE
      TEMPSUM = 0.
      DO 10 I = I1, I2
```

```
                TEMPSUM = TEMPSUM + A(I)
10          CONTINUE
C
C   UPDATE THE SHARED CRITICAL DATA STRUCTURE "SUM"
            CALL M_LOCK()
            SUM = SUM + TEMPSUM
            CALL M_UNLOCK()
            CALL M_SYNC()
            RETURN
            END
```

CHAPTER 3

IMPLEMENTATION OF THE REVISED SIMPLEX METHOD

In Chapter 2, we reviewed the salient aspects concerning the use of novel computing hardware to solve problems more efficiently. Another equally important aspect in the efficient solution of a problem is software efficiency. This chapter considers the implementation of the revised simplex method (RSM). In §3.1, the revised simplex method is reviewed. In §3.2, the specifications for a robust implementation of the revised simplex method on serial computers are described. Finally, in §3.3, efforts to solve linear programs on parallel computers are surveyed.

3.1 Revised Simplex Method

Consider the linear program (LP) in standard form.

$$
\begin{array}{lllll}
 & \text{Min} & \mathbf{c}\,\mathbf{x} & & \\
 & \text{s.t.} & & & \\
\text{(LP1)} & & \mathbf{A}\,\mathbf{x} & = & \mathbf{b} \\
 & \mathbf{x} & \geq & \mathbf{0} &
\end{array}
$$

where **A** is m x n, **c** is 1 x n and **b** is m x 1. Let

$\mathbf{c}_B$ be the basic cost vector,

$\mathbf{a}_j$ be the jth column of $\mathbf{A}$,

$\mathbf{B}$ be the basis matrix,

$\mathbf{d}_j$ be the vector obtained by updating $\mathbf{a}_j$ with $\mathbf{B}$ ($\mathbf{d}_j = \mathbf{B}^{-1}\mathbf{a}_j$),

π be the current dual vector and

$\mathbf{x}^*$ be the current solution vector.

The Revised Simplex Method (RSM) can now be stated in the following steps

1. Form $\mathbf{c}_B$.

2. Solve the system $\pi\mathbf{B} = \mathbf{c}_B$ for π.

3. Compute the reduced cost for all nonbasic columns and pick the column with the most negative reduced cost to enter the basis. If none goto 7. In this case the problem is optimal if the current solution is feasible and infeasible otherwise.

4. Solve $\mathbf{B}\mathbf{d}_j = \mathbf{a}_j$ for $\mathbf{d}_j$.

5. Find the pivot row using the ratio test. If none then declare the problem unbounded and goto 7.

6. Goto 1.

7. STOP.

3.2 Revised Simplex Implementation on Serial Computers

Two features that are essential to an efficient implementation of the RSM are column-wise packing and eta factorization of the basis matrix.

Column-Wise Packing. In the column-wise packing scheme, only the nonzero elements of the constraint matrix are stored. Since the density of nonzero elements in large-scale problems is typically low (< 1% according to Beale (1971)) this scheme saves us considerable storage.

The data structure used to implement column-wise packing requires three arrays A, IA and LA. The array A, stores the nonzero elements of the data matrix. The element IA(i) gives the row position of A(i) in the data matrix. The element LA(j) gives the position in A of the first nonzero element of the jth column of the data matrix. Therefore, LA(j+1) -1 gives the position in A of the last nonzero element of the jth column of the data matrix. Whenever the jth column of the data matrix is required, it is constructed from the information in the above arrays. This construction process is known as *unpacking* and is done as follows.

(a) Initialize an array Y to the zero vector.

(b) Do i = LA(j) , LA(j+1)-1

Y(LA(i)) = A(i)

EndDo

Eta Factorization of the Basis. In eta factorization, the basis matrix is always stored as a product of eta matrices. At the kth iteration, the basis (say $\mathbf{B}_k$) is given by (3.1).

$$\mathbf{B}_k = \mathbf{F}_1 \ldots \mathbf{F}_k \tag{3.1}$$

Here each $\mathbf{F}_i$ $(1 \leq i \leq k)$ is called an *eta matrix.* An eta matrix differs from the identity matrix in only one column called the *eta column.*

Due to eta factorization (eqn 3.1), steps 2 and 4 of the revised simplex method

can be implemented as given by (3.2) and (3.3). Such an implementation is efficient, since both the steps become a sequence of vector products in which one of the vectors is sparse (cf. Murtagh (1981)).

$$\pi = c_B \, E_k \ldots E_1 \tag{3.2}$$

$$d_j = E_k \ldots E_1 \, a_j \tag{3.3}$$

Here $E_i = F_i^{-1}$ $(1 \leq i \leq k)$.

Typically, eta factorization of the basis is done using a hybrid of two methods : the PFI (Product Form of Inverse) scheme due to Dantzig and Orchard-Hays (1954) and the EFI (Elimination Form of Inverse) scheme due to Markowitz (1957). The hybrid scheme is described below. If the first iteration of the simplex method is started with the identity, then at the end of each simplex iteration, the nonzero elements of the current updated entering column (the eta) are *simply appended* to the eta file (file to store eta matrices) provided there is space available in the file to write the elements of the column. This scheme of appending the updated entering column to the eta file is the PFI scheme. It is a *time-effective* way to maintain the basis matrix as a product of eta matrices. However, if only the PFI scheme is used then as the number of iterations increases, the eta file grows longer and consequently the computing time per iteration increases. A point is eventually reached when it is desirable to *reinvert* the current basis. The reinversion of the basis is accomplished using the EFI scheme which is a variant of L-U factorization [Noble and Daniel (1980)]. The eta file generated by the reinversion procedure is sparse, but the procedure is time consuming.

The reinversion procedure consists of two stages. In the first stage, the basis is

permuted and brought to a form in which only a submatrix of the basis (called the "bump") needs to be factorized and in the second stage this bump is factorized. After the first stage, the basis has the form shown in figure 3.1. Here, in sections A and E pivots have been found on the main diagonal and all other nonzeros are below it. Section B is all zero. Section C, the bump, contains at least two nonzero elements in each row and column. Factorizing the bump (section C) is done along the lines prescribed by Hellerman and Rarick (1972)

The pseudo-code of an efficient implementation of the revised simplex method is given below.

Program RSM;

Procedure INVERT;

- Does LU factorization of the basis

Procedure FORMC;

- Forms the cost vector

Procedure BTRAN;

- Computes the dual vector

Procedure PRICE;

- Computes reduced cost of nonbasic columns and identifies the lowest reduced cost and the column yielding it.

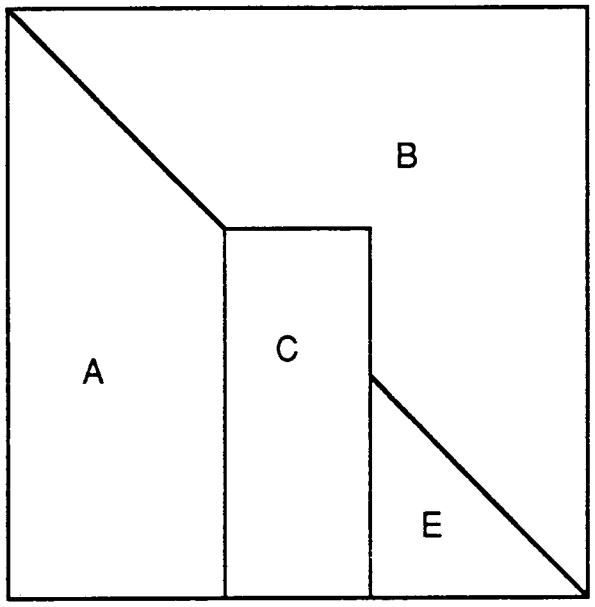


Figure 3.1 : Permuted basis during reinversion

Procedure FTRAN;

- Updates a column with the current basis

Procedure CHUZR;

- Finds the pivot row

Procedure UNPACK;

- Unpacks a column stored in standard column-wise packing

Procedure UPBETA;

- Updates solution after a basis change

Procedure WRETA;

- Writes to the eta file, the eta created after a basis change

Procedure INPUT;

- Reads the MPS file

Procedure ITEROP;

- Write iteration statistics

Procedure UNRAVL;

- Writes out the final solution

Procedure NORMAL;

1. INVERT
2. FORMC
3. BTRAN
4. PRICE
5. Determine entering column and if none goto 13
6. FTRAN
7. CHUZR
8. ITEROP
9. If no pivot row goto 13. Else effect basis change and if eta file is full goto 1
10. WRETA
11. UPBETA
12. Goto 2
13. STOP

Begin

1. INPUT
2. NORMAL
3. UNRAVL

End

3.3 Parallelization Efforts in Linear Programming

Research into parallel numerical optimization has been surveyed by Zenios (1989). In linear programming, parallelization efforts are progressing along two major directions. First, new decomposition methods and variants of existing methods (e.g. Dantzig and Wolfe (1960)) have been developed to solve linear programs on parallel computers. Since decomposition methods lend themselves naturally to parallel computation, the advent of distributed computers with several hundred processors has provided a new impetus to such methods. Second, several researchers have adapted the revised simplex method to vector computers (a class of SIMD computers; cf. §2.2). Many of the tasks of the revised simplex method require vector products and hence, this approach has speeded up the solution of linear programs significantly. In this section we survey work related to the parallelization of linear programming.

Forrest and Tomlin (1988) presented results for the IBM 3090 relating to vectorizing various steps in the simplex algorithm. Over a series of 13 test problems they reported that vector processing reduced execution time by a factor of 3.57. Stone, Koltz and Strauss (1989) discussed the vectorization of the simplex pricing operation on the IBM 3090. Over a series of 11 test problems they reported that vector processing reduced the excution time for pricing by a factor of up to 20. Bixby (1989) presented computational experience with the simplex algorithm on the Cray supercomputer. This simplex code is reported to be faster than others on Cray supercomputers. Helgason, Kennington and Zaki (1988) designed a parallel simplex algorithm. Instead of using the conventional eta factorization method to maintain the basis, their algorithm featured an alternative method, known as quadrant interlocking. Computational results for their algorithm are not available. There are a number of

papers wherein LPs are solved as part of a larger procedure (eg. Branch and Bound for Integer Programming). The reader is referred to Beasley (1989) or Zenios (1989) for a survey of this literature.

Some researchers have solved *structured* linear programs using decomposition based techniques. In general, the results validate the adaptability of decomposition techniques to distributed computers. Medhi (1987) developed a decomposition algorithm to solve a class of structured large-scale convex optimization problems, of which the block-angular linear program is a special case. He implemented a parallel version of this algorithm on CRYSTAL [DeWitt et al (1984)] and reported efficiencies of more than 70% with block-angular linear programming test problems when running up to 11 processors. The largest instances solved contained about 1500 rows, 4000 columns and 20 coupling constraints. Ho, Lee and Sundarraj (1988) (see also Lee (1988)) implemented, on CRYSTAL, a parallel version of the Dantzig-Wolfe decomposition algorithm [Dantzig and Wolfe (1960)] for block-angular linear programs. They solved a set of ten test problems the largest of which had 700 rows and 1250 columns. They reported speedups of up to about 8 and efficiencies of nearly 80 % when using up to 11 processors. Ho and Gnanendran (1989) implemented a parallel version (called DECUBE) of the Dantzig-Wolfe algorithm on the iPSC/2, a distributed memory hypercube multicomputer. It featured many refinements made possible due to parallel computation, and aimed at keeping all the processors busy. With problems of about 30,000 rows they report efficiencies of about 64% and speedups of about 19 when using 31 processors. DECUBE has been reported to be four to five times faster in comparison with MPSX/370 version 2; a commercial-quality linear programming code running on the IBM 3090.

Research into the use of MIMD computers to solve *unstructured* linear programs is still at an early stage and is the topic of this dissertation.

CHAPTER 4

TIMING MODELS

A timing model for an algorithm is a mathematical characterization of how processing time is spent in the algorithm. The objective of developing timing models is to study the behavior of an algorithm. Consider an iterative algorithm for which the iteration times undergo a linear growth with, say, slope A and intercept B. Then, the timing model for the iterations is (A+Bx) where x is the iteration number. The total run time for this algorithm is the summation of (A+Bx) over the total number of iterations.

A timing model can be a useful tool in algorithmic development because it can predict the performance of algorithms. The prediction of an algorithm's performance is significant, especially with the advent of parallel computers. While in conventional computing, algorithmic design is essentially independent of machine architecture, the diversity of parallel computing architectures implies that an algorithm designed for one type of parallel computer may not perform well on an another type. Typically, the only way of knowing how well the algorithm performs is by actually coding and testing it. However, an accurate timing model can be used to predict the algorithm's potential performance. In chapter 7, we propose new algorithms for solving linear programs and use the timing models developed in this chapter to predict the performance of those algorithms.

Timing models can be used to help design efficient parallel programs. The efficiency of a parallel program improves if its work load is evenly balanced (load balancing) among the processes of the program. There are two types of load

balancing : static and dynamic. In static load balancing, the allocation of workload to processors is done by the programmer before run time and is not changed during the run. Since the workload of processors does not change, there maybe times during the run when the workload is not evenly balanced among the processors. In dynamic load balancing, the workload is evenly balanced during the execution of the program. The timing model developed for an algorithm can be used to make dynamic load balancing possible. During the run, the timing model can be periodically used to infer the total workload on-hand and an equal share of that total can then be allocated to each process. This way, the idle time of processes is cut down and the elapsed time of the program is reduced.

In this chapter we develop timing models for the revised simplex and its modules, BTRAN, PRICE, CHUZR and UPDATE. The models for CHUZR and UPDATE can *predict* the run times of these modules on a given problem even before the run. We call such models *predictive*. The model for BTRAN, on the other hand, can only explain the run time after solving the problem and we call such models *descriptive*. For PRICE, we provide models of both types; a descriptive model which works well on all the problems tested and a predictive one which tends to work well on sparse problems. Although the descriptive models can only explain the run time of the modules, the insight it provides into the behavior of the algorithm, can be used to solve problems efficiently. For instance, suppose we observe that for a given class of problems the eta vectors generated after reinversion are dense. Then, for the solution of problems in this class we must reinvert frequently. With predictive models, we have the added advantage of obtaining such insights without having to solve the problem.

The chapter is organized as follows. In §4.2 we describe the model for the revised simplex method (RSM) and in §§4.3 through 4.6 the models for the RSM

modules are described. In each of these sections, the model is presented after describing the key points that are pertinent to its development. The reader is referred to Ho and Sundarraj (1989) or Murtagh (1981) for further details about the modules.

4.1 Preliminaries

4.1.1 SLPM : The Implementation Used for Model Validation

SLPM is a serial implementation of the revised simplex method (RSM) on the iPSC/2. It is based on LPM1, a robust all-in-core implementation of the RSM by Tomlin (1973). It runs on a d-0 cube (1 node) and consists of two programs, *host* and *node*, running on the host and node machines respectively. The host program reads in the data and passes it to the node which then solves the problem and passes back the results to the host. The pseudo-code for SLPM is given below.

host :

1. INPUT /* Read data from the MPS file. */
2. Load the node programs
3. CSEND the data
4. Wait for termination
5. Kill all processes

node :

1. CRECV the data
2. NORMAL (cf. §3.2 for details) /* All the timing is done here */

3. Send the results to the host.

The times reported in this chapter were obtained using the iPSC/2 routine MCLOCK(), which returns the elapsed time in milliseconds since the start of the process.

4.1.2 Performance Measure

The objective of developing timing models is to study the behavior of algorithms, especially its behavior on problems which take a long time to solve. Suppose one wants to evaluate the performance of a timing model for an algorithm. First, the algorithm is run on a set of test problems and the actual run times are obtained. Using the timing model, the run times are inferred and those inferences are compared with the actual run times. If the inferences are more accurate on certain problems than others, then the question is : "How can one assess the model's overall performance?" The following example illustrates how this question is to be addressed.

Consider the execution of an algorithm on a set of four test problems. Suppose one wants to evaluate two timing models for this algorithm. First, the algorithm is run on these problems. The actual run times are given in column 2 of the table below. Then, the timing model is used to infer the run times and these inferences appear in columns 3 and 5. The errors in the inferences versus the actual times are given in columns 4 and 6 and have also been graphed in figure 4.1. From the graph, we observe that model 1 performs better than model 2 on difficult problems (greater elapsed time) and should therefore be considered the better of the two

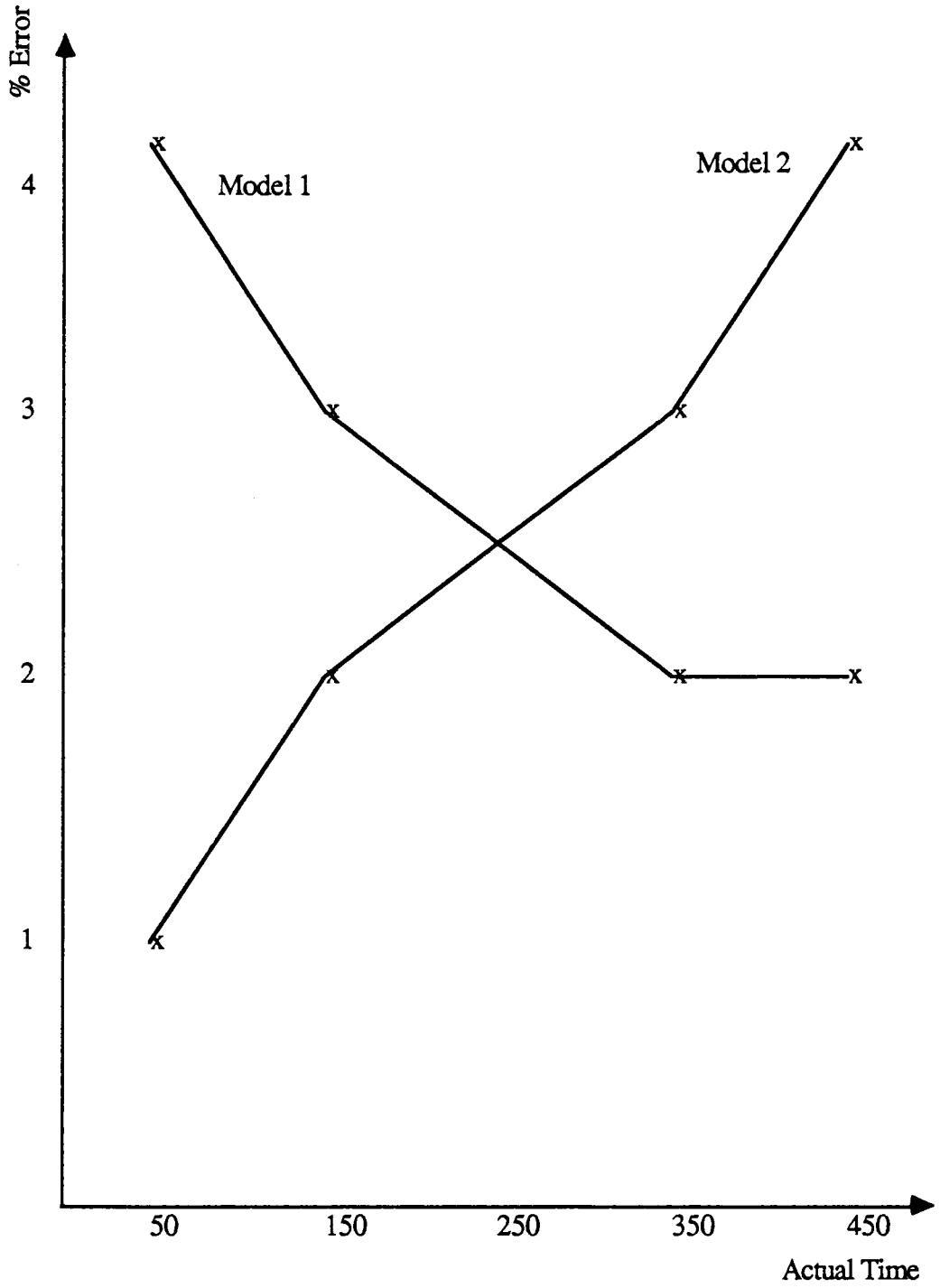


Figure 4.1 : Illustration of the models' errors

models for difficult problems. The average of the errors provides exactly the opposite conclusion. The average error does not taken into account the relative importance of the errors. However, if we weigh the errors by the elapsed time taken by the algorithm and use the average of the weighted errors then we would reach the correct conclusion.

Problem	Actual	Model 1		Model 2	
		Inference 1	% Error 1	Inference 2	% Error 2
1	50	48	4	49.5	1
2	150	145.5	3	147	2
3	350	343	2	339.5	3
4	450	441	2	432	4
Average			2.75		2.5
Weighted average			2.25		3.2

For a sample of p problems, the weighted average of a parameter x with respect to the parameter s is defined as follows. Let T(s, i) and T(x, i) [$1 \leq i \leq k$] be the values of parameters s and x respectively for the ith problem. Let SUM_s be the sum of T(s, i) for all i from 1 to p. The weighted average is then given by (4.1).

$$\sum_{i=1}^{p} [\, T(x, i) * (T(s, i) / SUM_s) \,] \qquad (4.1)$$

4.1.3 Partially Sparse Inner Product

Two major subroutines, BTRAN and PRICE, of the revised simplex method discussed in §3.2 make use of a Partially Sparse Inner Product (PSIP). We analyze

the algorithm for performing this inner product in order to develop the timing models for BTRAN and PRICE.

A PSIP is an inner product of two vectors one of which is stored in sparse form (cf. §3.1) while the other is stored in dense form (zeros are stored). Let a vector be stored in the array A in sparse form and let another vector be stored in the array Y in dense form. The pseudo-code to perform a PSIP is given below (Tomlin (1973)).

```
1. LL  <--      Start of the sparse vector in the array A
2. KK  <--      End of the sparse vector in the array A
3. INNR_PROD = 0.
4. For  PTR = LL, KK
       a. IR   <--      Infer the row position of A(PTR)
       b. INNR_PROD = INNR_PROD + Y(IR)*A(PTR)
   EndFor
```

Observe that step 4 is the *main part* of the algorithm. The time for executing this step depends only on the density of the sparse vector and this makes the algorithm very efficient. The *initialization steps* (steps 1 to 3) consist of one real assignment and two integer assignments.

4.2 Modeling the Revised Simplex Method

4.2.1 Revised Simplex Model

As discussed in §3.2, an efficient implementation of the revised simplex method uses a combination of two common schemes, the Elimination Form of Inverse (EFI) and the Product Form of Inverse (PFI), to maintain the basis matrix as a product of eta factors. The EFI scheme, due to Markowitz (1957), produces a sparse eta file but is time consuming. It is therefore used only to *reinvert* the basis (Hellerman and Rarick (1972)). Between reinversions, the PFI scheme is used and in that scheme the updated entering column is *simply appended* to the eta file at the end of each simplex iteration. The appending is easy to perform, but since the updated entering column is usually dense, the PFI scheme results in a dense eta file.

To describe the model for the revised simplex method, we define the term INVERT cycle. An INVERT cycle consists of executing an INVERT (the reinversion procedure) and the simplex iterations until the next call to INVERT. The number of iterations to be executed between two successive INVERT's *(inversion frequency)* is determined by the user. See figure 4.2 for an illustration.

Since the EFI scheme is used in INVERT, the size of the eta file decreases after INVERT is called. Subsequently, the eta file grows during the remainder of the INVERT cycle when the PFI scheme is used (see figure 4.3). This growth implies that in each successive iteration of an INVERT cycle, the time for the revised simplex subroutines BTRAN and FTRAN increases. To model this increase, we conducted an experiment that produces as many data points as the number of iterations in an INVERT cycle. The ith data point is the average of the times for the

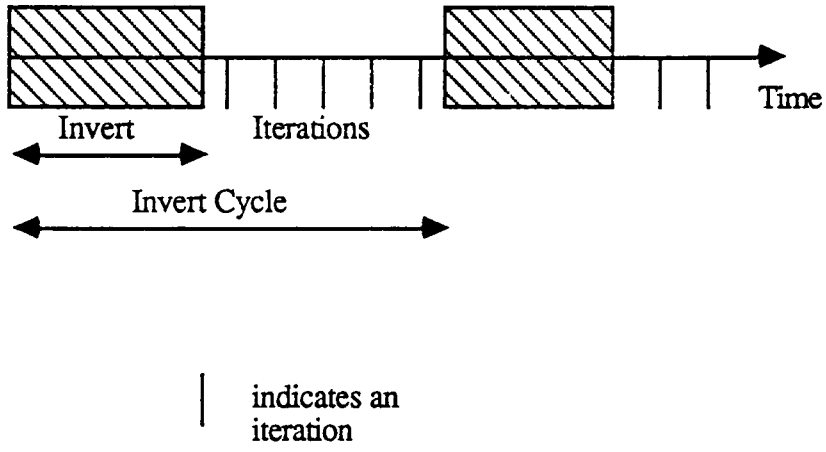


Figure 4.2 : Definition of INVERT cycle

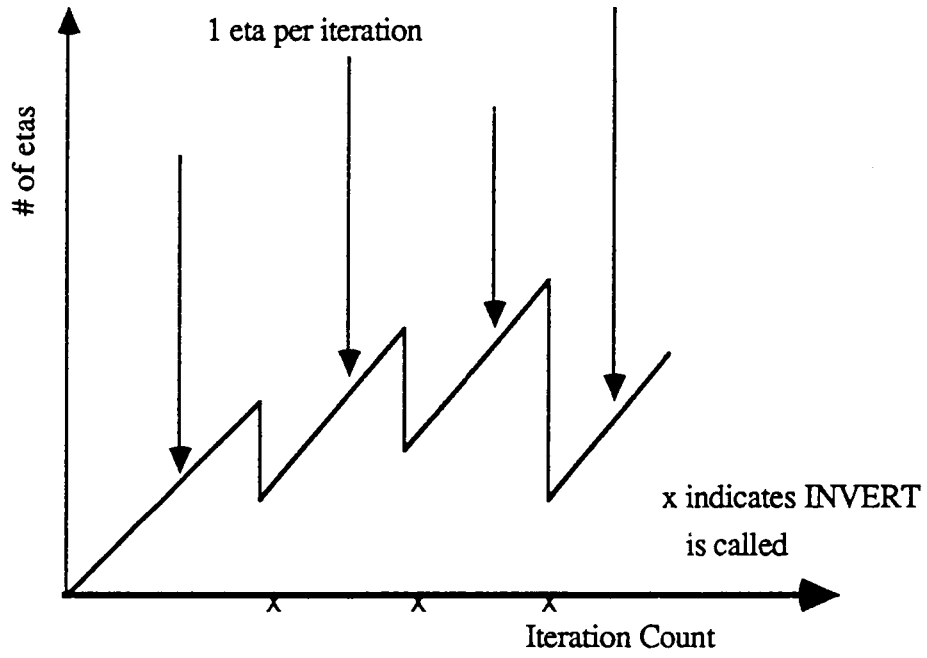


Figure 4.3 : Growth pattern of the etas in the revised simplex method

ith iteration after each INVERT. For example, suppose it takes 140 iterations to solve a problem using an inversion frequency of 50. Then, we would have a total of 50 data points and the fifth point (say) is the average of the times for the 5th, the 55th, and the 105th iterations. We conducted this experiment on a subset of our test problems. The plot of a typical set of data points is shown in figure 4.4.

From the picture, it is clear that the times for the iterations within an INVERT cycle can be described by a *linear model* of the form :

$$A + B*x$$

where A and B are the intercept and slope, respectively, of the linear model and x is number of iterations performed since the latest call to INVERT. This model implies that, on the average, each successive iteration within an invert cycle takes B time units more than the preceding one. Thus, given the slope and intercept of the linear model, we can infer the total time taken by all the iterations in an INVERT cycle and thereby the total time for all the simplex iterations (= number of cycles*time per cycle). The run time for the revised simplex method is then the sum of the time for the simplex iterations and the time for the reinversions.

Let

K be the total number of iterations,

k be the inversion frequency,

t_0 be the average time for the first iteration after an INVERT,

t_1 be the slope of the iteration growth pattern,

I be the mean INVERT time and

f be the inversion factor defined by I/t_0

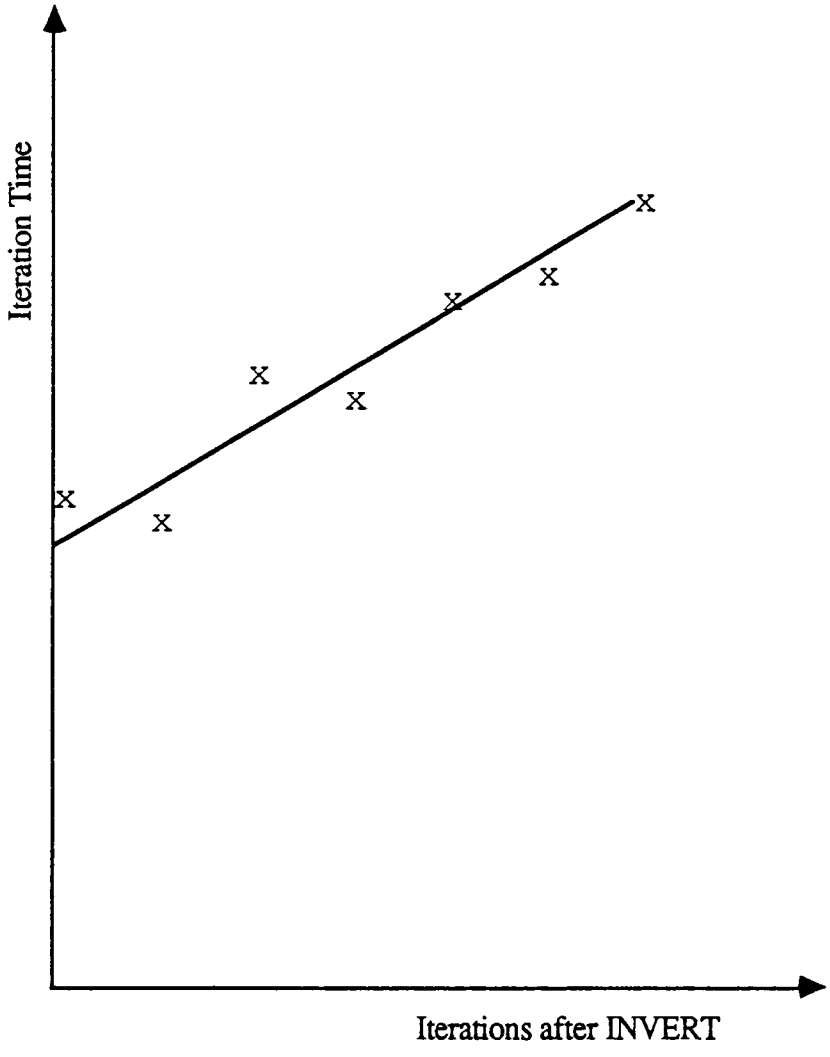


Figure 4.4 : Growth pattern of iteration time

The entire run consists of K/k INVERT cycles. During each INVERT cycle, an INVERT and k iterations are performed. The time for the ith iteration after an INVERT is $(t_0+t_1 i)$. Therefore, the total iteration time between consecutive INVERT's is the summation of $(t_0+t_1 i)$ over i from 1 to k. By substituting the INVERT time (I) with f^*t_0, the run time for the revised simplex method can now be stated mathematically as follows.

$$T_{SRSM} = (K/k)\ [\ f^*t_0 + \sum_{i=1}^{k} (t_0+t_1 i)\ ] \qquad (4.2)$$

$$= (K/k)\ [\ f^*t_0 + t_0 k + t_1 k(k+1)/2\ ]$$

$$T_{SRSM} = (K/k)^*f^*t_0 + Kt_0 + K^*t_1/2 + (k/2)^*K^*t_1 \qquad (4.3)$$

We found that the inferences of the run times, computed from (4.3), are close to the actual ones (columns 3 and 4 of Table 4.1). Over our set of test problems, the weighted average error (error = | inference-actual |) with respect to the actual run times is 0.64% (the unweighted error is 0.83%).

4.2.2 Economic Inversion Frequency

As stated in the above section, reinversion of the basis matrix is time consuming, but postponing reinversion implies that the eta file density increases and this inturn causes the iteration times to increase. An efficient implementation of the revised

Table 4.1 : Results with timing model for RSM

Problem	# of rows	Actual time in msecs	Inferred time in msecs
agg	489	16697	16826
agg2	517	22708	22368
agg3	517	24071	23822
bandm	306	189401	188938
brandy	221	88586	88424
israel	175	53108	53115
sc205	206	24303	24652
scagr25	472	159537	159839
scfxm1	331	65999	64981
scfxm2	661	283245	283415
scfxm3	991	736687	731786
scorpion	389	39969	39630
scrs8	491	203560	203273
scsd6	148	66382	66231
scsd8	398	399428	404463
sctap1	301	50461	50124
sctap2	1091	526241	523042
sctap3	1481	1126754	1123964
ship04l	403	145231	144045
ship04s	403	73887	73273
ship08s	779	195716	193248
ship12s	1152	462290	457277
zdeep1	101	55578	56183
zdeep2	101	40234	40931
zdynamico	678	1020908	1022257
zegd051	869	127708	126197
zegd085	1443	398422	392175
zfixmar	325	62072	61672
zforestry	402	197253	198275
zmeat12b	381	81004	80164
zmeat31b	384	88077	87168
zmeat43b	648	140301	138530
zmege06	1011	98568	97096
zmege08	1343	200714	199738
zmrp3	301	35847	35399

Mean absolute weighted error	0.64
Mean absolute error	0.83

simplex method must strike a balance between the cost of frequent reinversion and the cost of increased iteration time incurred by postponing the reinversion. A key factor to the efficient solution of a problem is the value of the inversion frequency (a user-supplied parameter). Equation (4.3) can be used to obtain an economic inversion frequency.Differentiating with respect to k, setting dT_{SRSM}/dk to 0 and simplifying that equation, gives :

$$k^* = SQRT(2f^*t_0/t_1) \qquad (4.4)$$

where SQRT(x) represents the square root of x. In order for the above equation to yield the economic inversion frequency, certain conditions must hold. First, the parameters in (4.3) [K, f, t_0, t_1] must be independent of k. Second, the number of iterations in every INVERT cycle must be equal to k*. The reason for requiring this condition is illustrated by the following example. If K = 140, f^*t_0 = 500, and t_1 = 0.4, then k* = 50 as per eqn (4.4). This implies that the optimal rate of reinversion is once every 50 iterations and this rate is maintained in every INVERT cycle, except the last. In the last cycle, reinversion is done to complete the remaining 40 iterations (=140 - 2*50) of the solution procedure. Thus, the last INVERT cycle will be truncated if K is not an integer multiple of k* and due to this truncation, k* is not the true optimal inversion frequency. In practice, since the number of iterations to solve a problem is likely to be large in comparison with the inversion frequency, the *truncation* of the last cycle will only have a *negligible* effect of the value of the economic inversion frequency.

We found that the run time for a problem decreases when k* is used as the inversion frequency (Table 4.2). The saving in the run time was measured as

Table 4.2 : Effect of economic inversion frequency on run time

Problem	# of rows	k	Time(k) in msecs	k*	Time(k*)	% saving
agg	489	50	122	25	117	3.6
agg2	517	50	142	43	131	8.0
agg3	517	50	147	37	138	5.8
bandm	306	50	175	37	172	1.8
brandy	221	50	137	27	127	7.7
israel	175	50	149	17	118	20.8
sc205	206	50	101	23	91	10.3
scagr25	472	50	203	36	192	5.4
scfxm1*	331	75	119	56	117	1.3
scfxm2	661	50	258	88	247	4.2
scfxm3	991	50	450	130	379	15.6
scorpion*	389	25	155	98	113	27.1
scrs8	491	50	205	62	202	1.7
scsd6	148	50	141	33	136	3.1
scsd8*	398	75	329	60	330	-0.2
sctap1	301	50	103	42	117	-13.9
sctap2	1091	50	365	90	350	4.2
sctap3	1481	50	579	122	514	11.2
ship04l	403	50	198	100	195	1.4
ship04s	403	50	152	105	149	2.4
ship08s	779	50	278	194	255	8.3
ship12s	1152	50	380	265	329	13.5
zdeep1	101	50	204	35	199	2.0
zdeep2	101	50	176	34	166	5.3
zdynamico	678	50	569	57	570	-0.3
zegd051	869	50	172	111	160	6.5
zegd085	1443	50	315	174	268	14.8
zfixmar*	325	25	128	49	118	7.6
zforestry	402	50	232	63	227	2.3
zmeat12b	381	50	146	44	146	0.1
zmeat31b	384	50	159	82	150	5.7
zmeat43b	648	50	215	183	176	17.8
zmege06	1011	50	239	135	228	4.5
zmege08	1343	50	340	215	308	9.5
zmrp3	301	50	86	55	85	1.0
			Mean weighted saving			7.18

follows. Each problem was run twice and the time per iteration of the two runs were compared. For the first run, we chose a standard value of 50 for the inversion frequency (denoted by k). This run provided us with the values of f, t_0 and t_1 which were then used to calculate k*. In some cases, such as SCFXM2, when the value of k* was close to 50, another inversion frequency k was arbitrarily chosen. The second run used k* as the inversion frequency. Let Time(x) denote the time per simplex iteration when x is used as the inversion frequency to solve the given problem. The run time saving for each problem, given by Time(k)-Time(k*), appear in Table 4.2 and the weighted average saving with respect to Time(k) is 7.18%.

From the table, it is seen that for many problems the run time savings is significant. Hence, if the value of k* can be inferred then one can solve problems more efficiently. Since for increasing problem sizes (more rows), the growth in the inversion time (numerator of (4.4)) is faster than the growth in the iteration time (denominator), we conjectured that the value of k* must increase with problem size. Initial results seem to support this conjecture. We see from Table 4.2 that for a given class of problems, k* tends to increase with problem size. For example, k* is 55.8 for SCFXM1, 88.4 for SCFXM and 129.5 for SCFXM3. Other classes include : AGGx, SCSDx, SCTAPx, SHIPxx, ZEGDxxx, and ZMEATxxx. Thus, for a given class of problems, one can parametrize the economic inversion frequency with problem size using carefully selected small problems that are representative of the class in terms of structure, density, and complexity. From this parametrization, the economic inversion frequency for a given problem can be extrapolated and used in its solution. This should be more effective than an arbitrary choice of k.

An interesting question is : "Is k* truly the optimum inversion frequency?" To answer this, we conducted an experiment consisting of a series of runs as explained below. First, we ran each problem using k* as the inversion frequency. This run

provided the values of f, t_0 and t_1 which were then used to calculate k_1*. Likewise, we then obtained k_2* using k_1*, k_3* using k_2*, and so on. The values of k, k*, k_1*, ... for a subset of our test problems appear in column 2 of Table 4.3. In many cases, such as AGGx, k* was nearly equal to k_i* ($i \geq 1$), whereas for SCORPION and SCFXM2 k* differed significantly from k_i*. For these two problems, the savings of k* Vs an arbitrary k was compared with the maximum of Time(k*)-Time(k_i*) ($i \geq 1$). That is, for SCORPION, we compared Time(25)-Time(98) with Time(98)-Time(85) and for SCFXM2 we compared Time(25)-Time(84) with Time(84)-Time(132). See column 3 of Table 4.3. For both the problems we see that k* is not truly the optimum inversion frequency, but the run time savings obtained using k* is much more than the extra savings provided by k_i*.

We summarize this section with the following remarks : (a) the growth of the iteration times within an INVERT cycle can be described by a linear model; and (b) Using the linear model we can obtain an economic inversion frequency.

4.3 Model for Backward Transformation

4.3.1 Backward Transformation Procedure

BTRAN, short for backward transformation, is the procedure for computing the dual vector. Mathematically, the dual vector, π is computed by solving the system

Table 4.3 : Effect of nonoptimality of economic inversion frequency

Problem	Sequence of economic inversion frequency	Comparison of savings
agg2	{25, 43, 46, 45}	
agg3	{25, 40, 38, 39}	
zfixmar	{25, 49, 50, 49}	
zforestry	{25, 66, 77, 77}	
zmeat 12	{25, 42, 41, 41}	
sc205	{25, 23, 22, 21, 22}	
sctap1	{25, 40, 41, 42, 40}	
scorpion	{25, 98, 85, 91, 96, 85}	27 Vs 8
scfxm2	{25, 84, 103, 110, 120, 132, 111}	25 Vs 4

$\pi \mathbf{B} = \mathbf{c}_B$ where $\mathbf{B}$ is the basis matrix and $\mathbf{c}_B$ is the vector of cost coefficients corresponding to the basic variables. From the above statement, the dual vector is given by :

$$\pi = \mathbf{c}_B \mathbf{B}^{-1} \tag{4.5}$$

In robust revised simplex codes, the basis inverse is represented as a product of eta matrices $\mathbf{E}_1, ..., \mathbf{E}_q$, as given by (4.6). Substituting (4.6) into (4.5) gives us, in eqn (4.7), the scheme to compute the dual vector.

$$\mathbf{B}^{-1} = \mathbf{E}_q \ ... \ \mathbf{E}_1 \tag{4.6}$$

$$\pi = \mathbf{c}_B \mathbf{E}_q \ ... \ \mathbf{E}_1 \tag{4.7}$$

In (4.7), etas generated by INVERT (EFI scheme) are called EFI etas and etas generated by the PFI scheme are called PFI etas. The number of eta transformations performed in a BTRAN is equal to the sum of the number of EFI etas generated by the latest call to INVERT and the number of PFI etas generated subsequent to that call (see figure 4.5). Without loss of generality, let us consider the transformation by the ith eta matrix, $\mathbf{E}_i$. At this stage, suppose $\mathbf{c}_B$ has already been transformed into, say, $\mathbf{c}'_B$ by the eta vectors $\mathbf{E}_q, ..., \mathbf{E}_{i+1}$. Recall (cf. §3.2) that an eta matrix is an identity matrix except for one nontrivial column, called the eta column (or the eta vector). That is, except for, say, its jth column which is the eta vector, all the other columns of $\mathbf{E}_i$ are unit column vectors. Therefore, the transformation of $\mathbf{c}_B$ by $\mathbf{E}_i$ would *only change* $(\mathbf{c}'_B)_j$ and this change is given by :

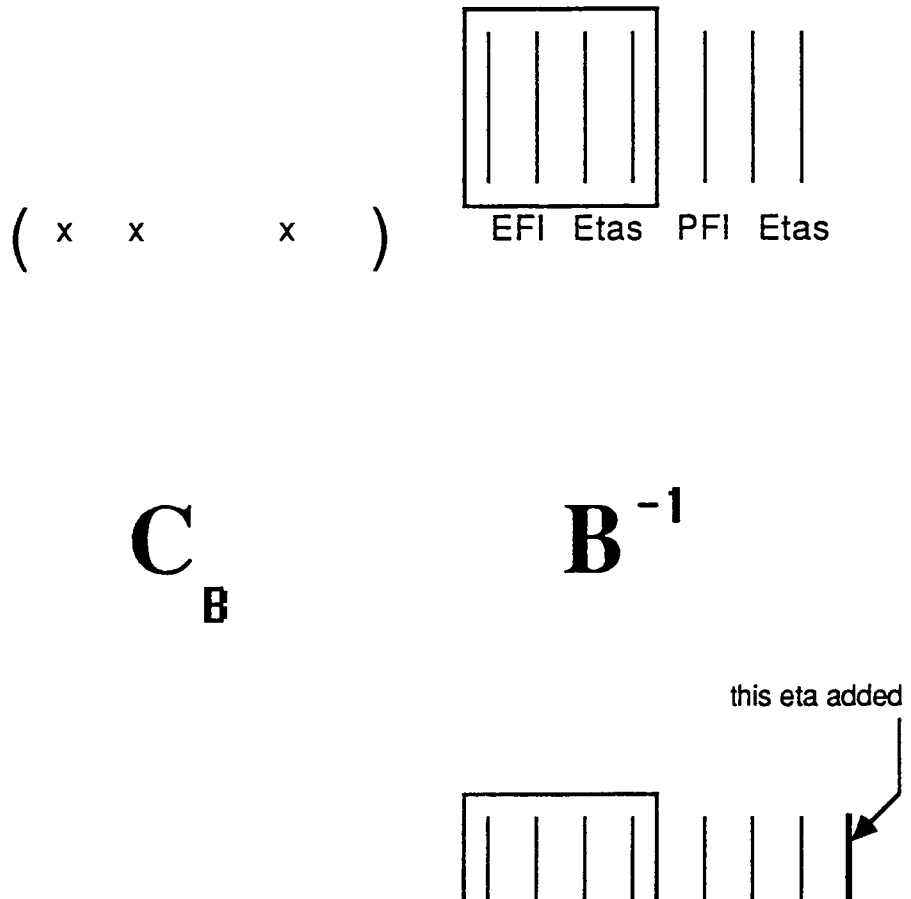


Figure 4.5 : BTRAN at consecutive iterations

$$[\, R^{*}e_{jj} \; + \sum_{k \neq j} (c_B)_k \, e_{kj} \,] \qquad (4.8)$$

where R is $(c'_B)_j$ and e_{*j} is the *th element of the eta vector.

From (4.8), it is seen that the main step of each BTRAN is the computation of the second term. This term, which shall be denoted by S, is the inner product of c_B and the eta vector, minus $(c'_B)_j e_{jj}$. Since eta vectors are stored in sparse form (only nonzeros stored), this inner product is a PSIP (Partially Sparse Inner Product; cf. 4.1.3) and therefore, BTRAN is essentially a sequence of q PSIP's.

The secondary steps of BTRAN are : (a) setting the loop index so that the eta transformation process goes from the last eta to the first eta; (b) inferring the pivot position j; (c) performing the initialization steps for a PSIP (steps 1 to 3 of §4.1.3); (d) skipping the loop to compute S, if the current eta vector is a singleton; and (e) excluding the $(c'_B)_j$ and e_{jj} from the inner product of c'_B and the eta vector. We can now state the pseudo-code for BTRAN as follows :

Procedure BTRAN :

For each eta i = 1, q

1Set eta index so that the transformation is started with the last eta.

2. Obtain j, the pivot position of the eta.

3. Do initialization steps of PSIP.

4. If eta is a singleton go to (7)

5. Exclude $(c_B)_j$ and e_{jj} from the computation of S

6. Perform the main step of PSIP to obtain S.

7. Evaluate (4.8) and assign the result to $(c'_B)_j$

End For

4.3.1 A Growth Model for BTRAN

Every BTRAN in an INVERT cycle performs the same number of EFI-eta transformations, but the number of PFI eta transformations increases by one for each successive BTRAN (see figure 4.5). Therefore, if A represents the time to transform the cost vector by the EFI etas and if B represents the transformation time per PFI eta, then BTRAN's time within an INVERT cycle can be described by a linear model of the form : A+Bx where x is the number of iterations that have elapsed since the latest call to INVERT.

We introduce the following notation in addition to the ones defined in the previous section. Let

r be the number of rows,

c be the number of structural columns,

n be the number of nonzeros in the structural columns,

e be the average number of EFI etas after an INVERT,

δ_e be the average number of nonzeros in an EFI eta, after an INVERT,

δ_p be the number of nonzeros per PFI eta,

t_{ip} be the time for the initialization steps of a PSIP (cf. §4.1.3),

t_{bp} be the time for secondary steps of BTRAN (steps 1,2,4 and 5) and

t_p be the time for performing the main part of the PSIP (step 4 of §4.1.3) of two real-numbered 1-dimensional vectors.

We can now derive the model for BTRAN. The entire run consists of K/k INVERT cycles, each containing k BTRAN's. At the ith iteration of an INVERT cycle ($1 \leq i \leq k$), e EFI eta transformations and i PFI eta transformations are performed. Steps 1 thru 5 take $t_{ip}+t_{bp}$ time units per eta transformation. Since step 6 is a PSIP, its time depends on the density of the eta vector involved in the transformation. Specifically, the time for step 6 depends on δ_e for the transformation by a EFI eta and on δ_p for the transformation by a PFI eta. Thus, the serial run-time for BTRAN (T_{SBTRAN}) is given by (4.9) which after simplification yields the final model as given in (4.10).

$$K/k * [\sum_{i=1}^{k} \{ e(\delta_e t_p + t_{ip} + t_{bp}) + i*(\delta_p t_p + t_{ip} + t_{bp}) \}] \quad (4.9)$$

$$K * [e(\delta_e t_p + t_{ip} + t_{bp}) + 0.5*(k+1)*(\delta_p t_p + t_{ip} + t_{bp})] \quad (4.10)$$

4.3.3 Results with Model for BTRAN

Using (4.10), T_{SBTRAN} can be computed if the values of the parameters δ_p, δ_e and e are known. The parameters δ_e and e depend on the results of INVERT. To obtain δ_e, the sum of the number of nonzeros in the eta file after each INVERT is divided by the sum of the number of etas after each INVERT. The parameter e is obtained similarly, by dividing the sum of the number of etas after each INVERT by the number of INVERT's performed.

Two methods were used to obtain δ_p and the first is as follows. At the end of the ith

simplex iteration ($1 \le i \le K$), we compute the number of PFI elements per PFI eta. Let this number be denoted by NE(i). The parameter δ_p is the average of : NE(1), ..., NE(K). For example, suppose it takes three iterations to solve a certain problem. and the number of PFI elements are 4, 8 and 21 after the first, second and third iterations respectively. Then, NE(1) is 4 (=4/1), NE(2) is 4 (=8/2), NE(3) is 7 (=21/3) and the parameter δ_p is 5 (= (4+4+7)/3).

The second method to obtain δ_p, is similar in idea to the one used in §4.2 for modeling the growth pattern of the iteration times. We obtain k data points, one for each BTRAN of an INVERT cycle. The ith data point is the average number of eta elements added by the ith iteration of an INVERT cycle. The value of δ_p is the average of these k points. For example, suppose it takes 140 iterations to solve a problem using 50 as the inversion frequency. We would then obtain 50 data points and the parameter δ_p is the average of these 50 points. For instance, the 5th point is the average number of nonzeros added by the 5th iteration of an INVERT cycle; that is, the average of the numbers of nonzero elements added by the 5th, the 55th and the 105th iterations.

An interesting side result follows from the data collected for this method. We expected each successive PFI eta to be denser than the preceding one and thus contribute to the increase in BTRAN's time. Table 4.4 shows the average density of the ith PFI eta ($1 \le i \le k$) of an INVERT cycle for the test problem SCFXM2. These densities are representative of our problems in the sense that there is no discernible pattern to the densities. We therefore conclude that the growth in BTRAN's time within an INVERT cycle is purely due to the increase in the number of etas, which justifies the use of a linear model for BTRAN.

The inferred times for BTRAN (T_{SBTRAN}) appear in Table 4.5. The weighted

Table 4.4 : Densities of PFI etas

Iterations after INVERT	Density of PFI etas
1	42
6	37
11	24
16	54
21	41
26	38
31	51
36	68
41	48
46	53

Table 4.5 : Results with timing model for BTRAN

Problem	# of rows	Actual time in msecs	Inference by method 1	Inference by method 2
agg	489	5151	4971	5435
agg2	517	5638	5009	5084
agg3	517	6369	5519	5735
bandm	306	70554	70489	70433
brandy	221	31421	30850	31050
israel	175	23518	23346	24227
sc205	206	10894	10986	11748
scagr25	472	61511	59907	61389
scfxm1	331	16220	16702	16376
scfxm2	661	55296	54356	55624
scfxm3	991	116253	117190	117600
scorpion	389	10868	9128	9632
scrs8	491	47742	47258	47238
scsd6	148	12166	12424	12388
scsd8	398	90258	89771	98969
sctap1	301	14911	12148	13133
sctap2	1091	92135	92686	94375
sctap3	1481	171276	172879	173869
ship04l	403	11026	10830	10935
ship04s	403	6554	6604	6647
ship08s	779	14229	15231	15231
ship12s	1152	34788	35556	35725
zdeep1	101	20161	21714	21883
zdeep2	101	15367	16622	16845
zdynamic	678	363363	348224	348218
zegd051	869	26715	27331	27521
zegd085	1443	71820	74296	74527
zfixmar	325	16135	17006	17226
zforestry	402	56312	55275	58221
zmeat12b	381	25364	25488	25843
zmeat31b	384	16829	17325	17696
zmeat43b	648	23342	24323	24475
zmege06	1011	14279	15676	15711
zmege08	1343	27891	28419	28486
zmrp3	301	7528	8254	8773
Mean absolute weighted error			2.74	3.48
Mean weighted error			4.37	4.92

average of the errors with respect to the actual BTRAN times is 2.74% for the first method and 3.48% for the second method.

4.4 Model for Pricing

4.4.1 Pricing Procedure

PRICE is the module that identifies the column with the minimum reduced cost (we assume a minimizing objective function). Mathematically, the reduced cost of the jth column is $c_j - \pi a_j$ where a_j is the jth column of the original coefficient matrix, c_j is the cost coefficient of jth column, and π is the dual vector. Ignoring the time to subtract πa_j from c_j, since it is negligible, pricing a nonbasic column is just the inner product of the dual vector and the nonbasic column, a_j. Since a_j is stored in sparse form, this inner product is a PSIP (Partially Sparse Inner Product). The pseudo-code for PRICE can now be stated as follows.

Procedure PRICE :

For each column i = 1 , c+r /* c structurals and r rows */

1. Check if column i is nonbasic and if not go to (6)
2. Do the initialization steps for a PSIP
3. Do main step of PSIP. This yields the reduced cost of column i
4. If (reduced cost of column i ≥ min reduced cost) goto (6)
5. Update minimum reduced cost

6. CONTINUE

End For

4.4.2 PRICE Model

As indicated earlier, pricing is essentially a sequence of PSIP's. Since the time for a PSIP depends on the density of the sparse vector (cf. §4.1.3), the *model* for the time to price a column is a function of the density of that column.

In addition to the notations defined in the previous sections we introduce the following. Let

δ_n be the number of nonzeros per nonbasic column,

t_{pp} be the time to do step 1,

t_{rc} be the time to do a real comparison,

t_{ra} be the time to perform a real assignment,

t_{ic} be the time required to do an integer comparison and

t_{ia} be the time for an integer assignment.

PRICE is called K times during a run, once per iteration. Step 1 is done c+r times and requires t_{pp} time units. Steps 2 thru 4 are each done c times. Step 2 takes t_{ip} time units and the number of calculations for step 3 depends on δ_n, the average number of nonzeros per nonbasic column. Step 4 involves a real comparison. The update step is performed c-1 times if the entering column (the column with the min reduced cost) is the last one in the list of nonbasic columns. It is performed once if

the entering column is the first in that list and is done c/2 times on the average. Thus, the serial run time for PRICE (T_{SPRICE}) by (4.11).

$$K * [(c+r)t_{pp} + c *(t_{ip} + \delta_n t_p + t_{rc}) + (c/2)*(t_{ra}+t_{ia})] \qquad (4.11)$$

Using (4.11), the value of T_{SPRICE} for a given problem can be computed if δ_n is known. At a given simplex iteration, let l : s denote the logical and structural column composition of the set of nonbasic columns. That is, at that simplex iteration, there must be lc/(l+s) logicals and sc/(l+s) structurals in the nonbasic set (note there are c nonbasic columns). Since there is exactly one nonzero in each logical column and since there are δ_s nonzeros per structural column, the total number of nonzeros in the nonbasic columns is given by (4.12). Hence, the number of nonzeros per nonbasic column is given by (4.13).

$$lc/(l+s) + \delta_s * sc/(l+s) \qquad (4.12)$$

$$\delta_n = (l+s*\delta_n) / (l+s) \qquad (4.13)$$

4.4.3 Results with Model for PRICE

To compute δ_n, the value of l : s is required and we propose two methods to infer it. The first method is *predictive* and it assumes that the logical and structural composition of the original constraint matrix (r : c) is maintained in the basis of every iteration. This implies that the composition of the set of nonbasic columns is also

r : c. That is, l : s is r : c. In the second method, we infer the ratio l : s after solving the given problem (*descriptive* method). From the run we obtain the total number of logicals and the total number of structurals in all the bases. Let them be x and y respectively. Then, the proportion of logical columns in a basis matrix is x/(x+y) and the proportion of structurals is y/(x+y). This in turn implies that out of a total of r logical columns, xr/(x+y) are in the basis and the remaining (r-xr/(x+y)) are in the nonbasic set. Using a similar line of reasoning, we conclude that the number of structurals in the nonbasic set is c - yr/(x+y). That is, l : s is (r-xr/(x+y)) : (c-xr/(x+y)).

For our test problems, we inferred the values of δ_n using both methods described above and from that we computed the inferred value of T_{SPRICE} using (4.11). For the descriptive model, the weighted average of the errors with respect to the actual times is 4.16% for our set of test problems. We observe that the inferences with this model are, in general, close to the actual value of T_{SPRICE} (see column 6 of Table 4.6). However, the predictive model performs well only on certain problems, since the prediction of δ_n is accurate only for some of the test problems. We now attempt to determine the cases for which the predictive model does well.

(a) Suppose we have an LP for which the range of the column densities (max δ_s - density of logical which is one) is small. Then, the actual value of δ_n will differ from its inferred value by only a small amount (< max δ_s - 1). Hence, in such cases, the inference of T_{SPRICE} will be accurate. For our set of problems, observe from column 5 of Table 4.6, that although there is no strict correlation between the problem sparsity and the accuracy with which the pricing time can be predicted, the

Table 4.6 : Results with timing model for PRICE

Problem	# of rows	Actual time in msecs	Structural column density	Inference of predicti-ve model	Inference of descripti-ve model	l : s for predictive model	l : s for descripti-ve model
sc205	206	1570	2.7	1923	1885	1.01	0.88
scrs8	491	51039	3.4	55842	56342	0.42	0.49
zmrp3	301	9309	3.9	10795	10946	0.58	0.67
ship12s	1152	191215	4.0	177147	195283	0.42	1.81
ship08s	779	98557	4.0	91452	99561	0.33	1.80
ship04s	403	41269	4.0	39166	41744	0.28	1.33
ship04l	403	93879	4.0	89112	93271	0.19	1.15
scagr25	472	13294	4.1	18970	18493	0.94	0.82
scsd8	398	201559	4.1	199542	200769	0.14	0.24
scsd6	148	39585	4.2	36160	36424	0.11	0.25
sctap1	301	12212	4.3	12415	13023	0.63	0.94
sctap2	1091	144716	4.3	145074	150078	0.58	0.80
sctap3	1481	257513	4.3	257610	266959	0.60	0.82
scorpion	389	5964	4.8	6768	6962	1.09	1.22
zegd051	869	13319	5.3	15224	17105	2.13	2.76
zegd085	1443	37440	5.3	43153	48394	2.12	2.74
bandm	306	21940	5.6	31986	30536	0.65	0.47
scfxm1	331	16077	5.7	15599	16136	0.72	0.89
scfxm2	661	64102	5.7	62227	64521	0.72	0.90
scfxm3	991	145177	5.7	139288	144731	0.72	0.91
zfixmar	325	14872	5.8	14534	15046	0.72	0.89
zforestry	402	36972	6.3	34428	36957	0.67	1.05
zmeat31b	384	27759	7.4	24266	26918	0.67	1.24
zmeat43b	648	31446	7.8	27745	31338	1.07	1.61
brandy	221	13342	8.6	12694	13237	0.89	1.04
zmege06	1011	30884	8.8	19873	31745	1.47	7.76
zmege08	1343	59110	9.0	38017	60363	1.48	7.16
zmeat12b	381	14091	9.6	13487	14222	1.23	1.40
zdynamic	678	78781	10.4	71351	70555	1.36	1.33
agg2	517	7919	15.0	4620	8097	1.71	6.82
agg3	517	8036	15.0	4748	8161	1.71	6.20
agg	489	3417	15.6	1781	3647	3.00	9.65
israel	175	5886	16.6	5914	6919	1.23	1.74
zdeep1	101	9979	30.4	11404	10246	0.62	0.36
zdeep2	101	5908	33.8	7289	6176	0.81	0.49

Mean absolute weighted error	7.47	4.27
Mean absolute error	14.79	8.18

predictions for problems with $\delta_S < 7$ are, in general, better than those for problemswith $\delta_S \geq 7$. That the inferences are, in general, better for sparser problems is also borne out by the fact that the weighted average error of the inferred times with respect to the actual pricing time is 4.73% when considering problems with $\delta_S < 7$, while it is 19.9% when considering problem with $\delta_S > 7$.

(b) For denser LP's, the inferences of the predictive and descriptive models were not always close to one another. Recall, that the models differ from one another only in the manner by which they obtain l : s (number of logicals : number of structurals in the nonbasic set). Therefore, the differences in the two inferences are significant whenever the two inferences of l : s are drastically different. See, for instance, AGGx and ZMEGE0x. The two inferences were close to each other otherwise. See for instance, MEAT12B and DYNAMICO.

4.5 Model for Pivot Row Selection

4.5.1 Pivot Row Selection Procedure

CHUZR is the module that identifies the leaving variable. The leaving variable is that variable whose nonnegativity imposes the most stringent threshold on the increase of the value of entering variable. In the conventional simplex method, since the right hand side is always positive, this threshold is obtained through the minimum (min for short) ratio test which consists of : (a) taking the ratio, $(\mathbf{x}^*)_i$ / $(\mathbf{d}_j)_i$ for all rows i with $(\mathbf{d}_j)_i > 0$ ($\mathbf{x}^*$ is the updated solution and $\mathbf{d}_j$ is the updated

entering column); (b) setting the threshold to the minimum of the ratios obtained above and (c) picking the basic variable of the row with the min ratio to leave the basis. The row which yields the min ratio is called the pivot row.

Computationally, two tasks are to be accomplished by CHUZR. The first is to set the threshold value. The second task, meant for numerical stability, is to select the row with the maximum pivot element from among those rows yielding the same threshold value. The row with the maximum pivot element is the pivot row.

Since the right hand side (rhs) in revised simplex codes is allowed to be negative, we can no longer restrict ourselves to "positive elements of $\mathbf{d}_j$ and the minimum ratio" to determine the threshold on the increase of the entering variable [Ho and Sundarraj (1989)]. Instead, there are now four cases contributing thresholds and corresponding candidate pivot rows, as summarized below. The fixing of the actual threshold (super-threshold) is not computationally significant and is therefore not discussed here.

Case	rhs and entering column coefficients		action
1	+	+	Threshold is the minimum ratio
2	-	-	Threshold is the maximum ratio
3	+	-	IGNORE
4	-	+	IGNORE

Among the four cases above, the number of computations in the first two cases is the same. In both those cases, the following ratio test is performed : (a) take ratio; (b) compare ratio with threshold value; and (c) update the threshold (in case 1we update if ratio is smaller than threshold and in case 2 we update if ratio is greater). In addition to the ratio test, the following steps are performed. To ensure numerical

stability, we perform the "pivot tolerance test" wherein we test if the element in the current row of the updated entering column is less than the user-supplied pivot tolerance (the ratio test is not performed if element is close to tolerance). Also, if the basic variable of a row is a nonbinding logical then it must never leave the basis. The checks done to ensure this consist of : (i) inferring the basic variable for the current row and (ii) checking if the row type is nonbinding in case the basic variable is a logical.

The time for finding the maximum pivot element (second task) is related to that for the first task (threshold fixing). Case 2 of the table above is further subdivided into two : (a) rows in which an artificial variable is basic and (b) rows with other types of basic variables. The maximum pivot element is determined only if case 1 or case 2b yielded the threshold (cf. Ho and Sundarraj (1989)). We can now state the pseudo-code for CHUZR as follows.

Procedure CHUZR

A. For each row i = 1, r /* Individual Thresholds are set */

1. If absolute value of $(d_j)_i$ < pivot tolerance goto A5 (pivot tol test)

2. Check if basic variable is a nonbinding logical and if so goto A5.

3. Decide on the CHUZR cases (cf. table above)

4. Perform ratio test (see above)

5. CONTINUE

End For

B. For each row i = 1, r /* Determine max pivot row */

1. Check if basic variable is a nonbinding logical and if so goto B7

2. Decide on the CHUZR case

3. Take ratio for cases 1 and 2b above

4. If ratio is not "better" than threshold then goto B7

5. Compare current pivot with max pivot element

6. Update max pivot if necessary

7. CONTINUE

End For

4.5.2 CHUZR Model

In addition to the notations defined in the previous sections, we introduce the following. While fixing the threshold, let f_1 be the fraction of rows for which the ratio test is performed (cf. CHUZR cases). While determining the max pivot, let f_2 be the fraction of rows for which the ratio is taken anf f_3 be the fraction of rows which yield the same threshold.

CHUZR is done K times, each consisting of two scans through r rows as follows. The first scan involves setting the threshold and is modeled as follows. Step A1 involves a real comparison. Step A2 consists of an integer assignment and two integer comparisons. Step A3 requires two real comparisons. Step A4, performed for a fraction (= f_1) of the rows, consists of a real multiplication, a real comparison and a real update. The update here is done just once if the threshold is provided by the last row. It is done f_1*r times if the last row yields the threshold and is done f_1*r/2 times, on the average. The second scan is for finding the max pivot element. Steps B1 consists of an integer assignment and two integer comparisons. Step B2 consists of two real comparisons. Step B3, performed for a

fraction (= f_2) of the rows, consists of a real multiplication. Step B4, performed for a fraction (= f_3) of the time the ratio is taken, requires a real comparison. Step B5 needs a real comparison. The update step (B6) requires a real assignment. Then the time, T_{SCHUZR}, taken for pivot row selection is given by (4.14).

$$K * [\ r * \{\ t_{rc} + t_{ia} + 2*t_{ic} + 2*t_{rc} + f_1*(t_{rm}+t_{rc}+.5*t_{ra})\ \}\ ]\ +$$

$$K * [\ r * \{\ t_{ia} + 2*t_{ic} + 2*t_{rc} + f_2*(t_{rm} + t_{rc}+ f_3*(t_{rc}+.5*t_{ra}))\ \} \qquad (4.14)$$

From (4.14), it is seen that the *key* parameters are f_1, f_2 and f_3 and we now describe how they were obtained. We assume that an equal distribution of positive and negative elements occur in the updated entering column as well as in the right hand side. While fixing the threshold, since the ratio is taken for two CHUZR-cases [(+,+) and (-,-)], f_1 is 0.5. While determining the max pivot element, recall that the second case [(-,-)] is further broken up into two and out of the resulting two cases the ratio is taken only for one case. Thus, we estimate f_2 to be 0.25 + 0.25*0.5 (=3/8). The parameter f_3 measures the number of rows yielding the same threshold. Except for highly degenerate problems, this number must be very small and we assume it to be 1%. The inference of T_{SCHUZR} appears in Table 4.7. We varied f_2 from .24 to .41 and f_3 from .005 to .02. In Table 4.8, the values of the mean absolute weighted error (1st number) and the mean weighted error (2nd number) are given for various values of f_2 and f_3.

Table 4.7 : Results with timing model for CHUZR

Problem	# of rows	Actual time in msecs	Inferred time
agg	489	3034	3066
agg2	517	3710	3848
agg3	517	3827	3944
bandm	306	15987	15213
brandy	221	7245	6586
israel	175	3602	2886
sc205	206	2484	2285
scagr25	472	17909	17149
scfxm1	331	8104	8392
scfxm2	661	31613	33391
scfxm3	991	69979	74680
scorpion	389	6174	6585
scrs8	491	21663	22243
scsd6	148	3081	2913
scsd8	398	21888	21892
sctap1	301	6615	6802
sctap2	1091	67582	72181
sctap3	1481	123481	132344
ship04l	403	12605	13444
ship04s	403	8313	8898
ship08s	779	23342	25004
ship12s	1152	59477	63975
zdeep1	101	1617	1260
zdeep2	101	1397	1058
zdynamico	678	59341	55804
zegd051	869	27867	29722
zegd085	1443	78258	83914
zfixmar	325	7479	7666
zforestry	402	15493	15594
zmeat12b	381	9025	9714
zmeat31b	384	8700	9737
zmeat43b	648	18099	19418
zmege06	1011	18090	19163
zmege08	1343	34308	36367
zmrp3	301	5416	5702

Table 4.8 : Weighted errors Vs values of parameters

		f3 values				
		0.0025	0.005	0.01	0.015	0.02
f2 values	0.24	3.49, 12	3.5, .11	3.51, .09	3.52, .07	3.53, .06
	0.25	3.64, -.14	3.64, -.15	3.65, -.17	3.66, -.19	3.67, -.21
	0.26	3.77, -.41	3.78, -.42	3.79, -.44	3.8, -.46	3.81, -.48
	0.29	4.2, -1.2	4.2, -1.2	4..22, -1.23	4.24, -1.25	4.25, -1.27
	0.32	4.67, -1.99	4.68, -1.99	4.7, -2.02	4.71, -2.05	4.73, -2.07
	0.35	5.18, -2.77	5.19, -2.79	5.21, -2.82	5.23, -2.84	5.25, -2.87
	0.38	5.72, -3.58	5.72, -3.58	5.74, -3.61	5.76, -3.64	5.77, -3.67
	0.41	6.25, -4.36	6.26, -4.37	6.28, -4.4	6.31, -4.44	6.33, -4.47

4.6 Model for Solution Updating

4.6.1 Solution Updating Procedure

This module updates the solution after a basis change. The ith element ($1 \le i \le r$) of the updated solution is given by (4.15).

$$(\mathbf{x}^*_{new})_i = \begin{cases} \beta & i = p \\ (\mathbf{x}^*_{old})_i - (\mathbf{d}_j)_i * \beta & i \neq p \end{cases} \qquad (4.15)$$

where p is the pivot row index , $\mathbf{x}^*_{new}$ is the updated solution vector, $\mathbf{x}^*_{old}$ is the old solution vector, β is the value at which the entering column comes in to the basis and $\mathbf{d}_j$ is the updated entering column.

Computationally, there is the potential for numerical instability when one of the elements of the solution vector is "close" to the zero tolerance of the machine. To avoid this instability, a check has to made to assign such elements to zero. The pseudo-code for UPDATE can now be stated as follows.

Procedure UPDATE

For each row i = 1,r

1. If pivot row go to (d)
2. Perform the update given by (4.16)
3. Perform the zero tolerance check stated above

4. CONTINUE

End For

4.6.2 Time Taken by UPDATE

Let t_{rs} be the time required to do a real addition/subtraction, and t_{rm} be the time for a real multiplication.

UPDATE is done K times. Step 1 is performed r times and involves an integer comparison. Steps 2 and 3 are each done for nearly r elements (precisely r-1). Step 2 involves a real addition and a real multiplication. Step 3 consists of a real comparison. Thus, the model for UPDATE is given by (4.16)

$$K * [\ r t_{ic} + (r-1)*(t_{rs}+t_{rm}+t_{rc})\ ] \qquad (4.16)$$

Given the value of K, this model is predictive, since all other parameters of the model are known a priori. The results of this predicton appear in Table 4.9.

Table 4.8 : Results with timing model for UPDATE

Problem	# of rows	Actual time in msecs	Inferred time
agg	489	1272	1304
agg2	517	1585	1646
agg3	517	1620	1687
bandm	306	6623	6589
brandy	221	2974	2841
israel	175	1237	1240
sc205	206	1019	984
scagr25	472	7417	7392
scfxm1	331	3650	3629
scfxm2	661	14423	14442
scfxm3	991	32129	32301
scorpion	389	2902	2833
scrs8	491	9893	9683
scsd6	148	1351	1281
scsd8	398	9863	9615
sctap1	301	2891	2947
sctap2	1091	30567	31305
sctap3	1481	56316	57378
ship04l	403	5847	5895
ship04s	403	3835	3890
ship08s	779	10849	10913
ship12s	1152	27640	27852
zdeep1	101	561	549
zdeep2	101	464	460
zdynamico	678	23945	24204
zegd051	869	12332	12865
zegd085	1443	34782	36323
zfixmar	325	3379	3337
zforestry	402	6820	6792
zmeat12b	381	4256	4216
zmeat31b	384	4208	4241
zmeat43b	648	8464	8433
zmege06	1011	7924	8309
zmege08	1343	15045	15767
zmrp3	301	2367	2486

Mean absolute weighted error	1.99
Mean absolute error	2.17

CHAPTER 5

DISTRIBUTED ALGORITHMS

In this chapter, we derive distributed algorithms for the various tasks of the revised simplex method, such as reinversion (INVERT), backward transformation (BTRAN), pricing (PRICE), pivot row selection (CHUZR) and solution updating (UPDATE). The algorithms are named distributed INVERT, distributed PRICE and so on (§§5.2 thru 5.5). In the distributed INVERT and distributed BTRAN algorithms, we create multiple processes and have them execute different tasks simultaneously. These algorithms will be called *heterogenous* distributed algorithms. In each of the other algorithms, the same task is executed on the processors and therefore such algorithms are called *homogenous* distributed algorithms.

The description of our algorithms is organized as follows. First, we provide a generic statement of the algorithm without describing machine dependent aspects of its implementation. Next, we describe details concerning the implementation of the algorithm on distributed memory and shared memory computers. Using the timing models developed in chapter 4, this implementation's potential saving on the wall-clock time to solve a given problem is inferred. Inferred savings for distributed INVERT and distributed PRICE were significant and this led to their actual implementation on the iPSC/2. Finally, empirical results with these two algorithms are reported.

5.1 Modeling Interprocessor Communication Time

Algorithms designed for parallel computers typically require some degree of interprocessor communication. The speed of this communication varies drastically from one type of machine to another. Consider the Sequent Balance, a shared memory MIMD computer (cf. §2.4.2). Here, there is a common pool of memory accessible to all processors and interprocessor communication is achieved by setting the appropriate variables in this shared memory bank. The time for communicating, say, 8 bytes between two processors is essentially the time for executing a floating point assignment statement and this time is negligible. However, on a distributed memory MIMD computer (cf. §2.4.1) such as the iPSC/2, communication is done by passing messages via a communication network. Since these networks are slower than the components that perform floating point computations, communication time on the iPSC/2 is significant. Therefore, the analysis of an algorithm designed for the iPSC/2 must take this time into account. In this section we model the communication mechanism used by the iPSC/2.

We found the time to send messages of various lengths between two processors. For a given message length, twenty messages were sent between two processors and timed. Processor 1 sent a message of length LNTH to processor 2 which, on receipt of this message, sent back one of the same length. This was repeated ten times (thus 20 messages). The communication times measured, appear in Table 5.1 and have also been graphed in figure 5.1. From the figure, we see that the interprocessor communication time on the iPSC can be described by a linear model. Using linear regression, we found that the time to communicate a message between two processors is given by the trend line:

Table 5.1 : Communication times for the iPSC/2

Message length	Communication time
10	7.2
100	7.8
1000	21.6
10000	86
50000	373
100000	732
150000	1092
200000	1450
210000	1522
220000	1594
230000	1666

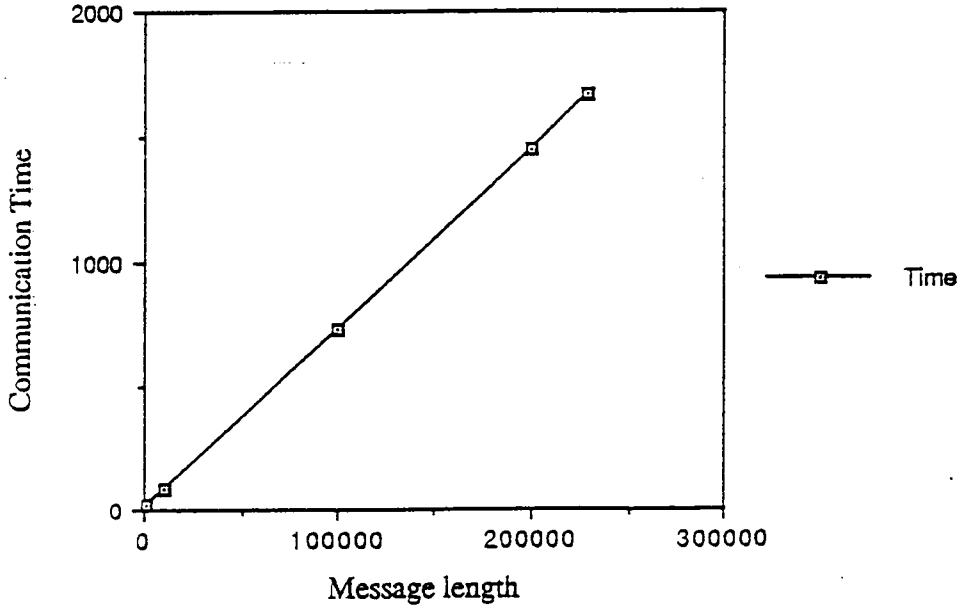


Figure 5.1 : Graph of communication times for the iPSC/2

11.3 + .0072 * (Message Length).

Each processor on the iPSC/2 has two areas of memory : the program area for storing the user's program and the system buffer area, which as explained below is used when performing interprocessor communication. The communication of a message between the program areas of two processors requires two steps. In the first step, the message must be transferred from the program area of the first processor to the system buffer of the second. To do this, the iPSC-supplied communication routine, CSEND (or ISEND) must be called from the user's program. In the second step, the message in the second processor's system buffer area must be transferred to its program area and this is done by calling the iPSC-supplied communication routine CRECV (or IRECV) from the user's program. We now model the time taken by these two steps.

The arguments of CSEND (or ISEND) are : the type of the message to be sent (TYP), the pointer to the memory location from where the message is to be sent (STRT), the length of the message (LNTH), the ID of the destination processor (DEST) and the ID of the process (PID). With CSEND, the following tasks are executed (see figure 5.2) : (1) starting from STRT, the sending processor copies LNTH bytes from the program's area to the system buffer and then returns to execute the program; (2) a *link* between the sending and destination processor is then established via the communication network and (3) the message is then relayed across that link to the system buffer of the destination processor. With ISEND, the sending processor simply posts a request to the operating system to send the message and then continues with the execution of the program. Step 1 involves copying information from one memory location (program area) to another (system buffer) and hence, the time for this step increases with the message length. Irrespective of the message length, a link has

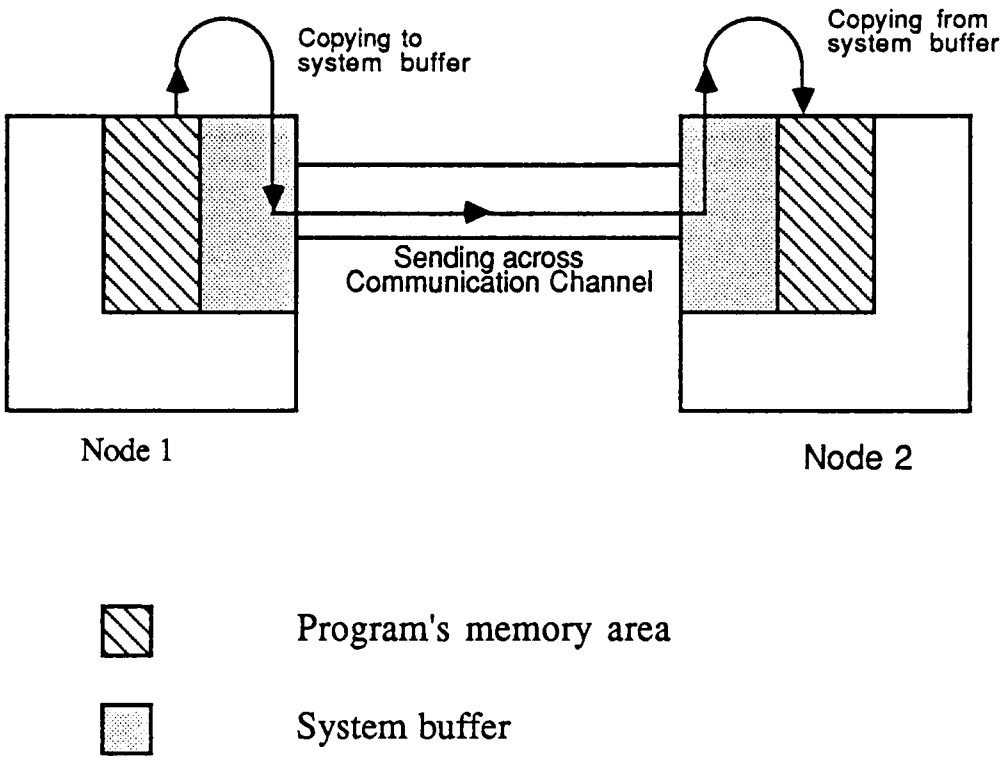


Figure 5.2 : Message sending and receiving process

to be established to relay a message and hence, the time for establishing the link represents the fixed cost of communicating a message. For step 3, the duration for which the link has to remain established increases with the message length. Let t_{lnk} be the time for establishing a link, t_{cop} be the time to copy a byte from the system buffer to the program area and t_{relay} be the time to relay a byte across the link. Hence, the time, T_{csend}, to send a message from the program area of one processor to the system buffer of another is given by (5.1).

$$T_{csend} = t_{lnk} + (t_{cop}+t_{relay})*LNTH \qquad (5.1)$$

The arguments of CRECV (or IRECV) are : the type of the message to be received (TYP), the maximum length available for the message (LNTH) and the pointer to the memory location where the received message is to be stored (STRT). With CRECV, the following steps are executed : (a) the processor running the program waits until a message of the type TYP arrives at its system buffer and (b) the message is then copied from the system buffer to the program area. With IRECV, the receiving processor simply posts a request to the operating system to receive the message and then continues with the execution of the program. The wait time (step 1) is dependent on the nature of the algorithm and therefore, cannot be modeled. The time for step 2, as in the CSEND case, increases with the message length. Thus, the time, T_{crecv}, to receive a LNTH-byte message is given by (5.2)

$$T_{crecv} = t_{cop}*LNTH \qquad (5.2)$$

5.2 Distributed INVERT

In the revised simplex method, the basis is reinverted to obtain a sparse eta file. Although frequent reinversion helps solve a problem efficiently, the continuation of the simplex iterations does not require the completion of the reinversion. Thus, after calling INVERT (the reinversion procedure), simplex iterations can be executed without waiting for INVERT to be completed. The Distributed INVERT (DINV) algorithm exploits this fact in a distributed computing environment.

5.2.1 Distributed INVERT Algorithm

The DINV algorithm uses two processors, denoted master and slave which work in parallel. The master is responsible for dispatching the slave to reinvert a given basis (say $\mathbf{B}_x$), whereas the slave just completes the reinversion of $\mathbf{B}_x$ and reports back to the master. In addition, the master executes the revised simplex subroutines.

A flowchart of the algorithm appears in figure 5.3. As shown in the figure, at the end of each simplex iteration, the master checks to see if the slave has completed its task. If the slave is still busy, the master executes another iteration. Thus, while the slave is reinverting $\mathbf{B}_x$, the simplex process on the master processor advances to a new basis, say $\mathbf{B}_y$ and PFI etas (etas added by the Product Form of Inverse scheme) are added to the eta file (see figure 5.4). As soon as the master detects that the slave has completed the reinversion of $\mathbf{B}_x$, it performs the following synchronization steps and then dispatches the slave to reinvert the new basis $\mathbf{B}_y$.

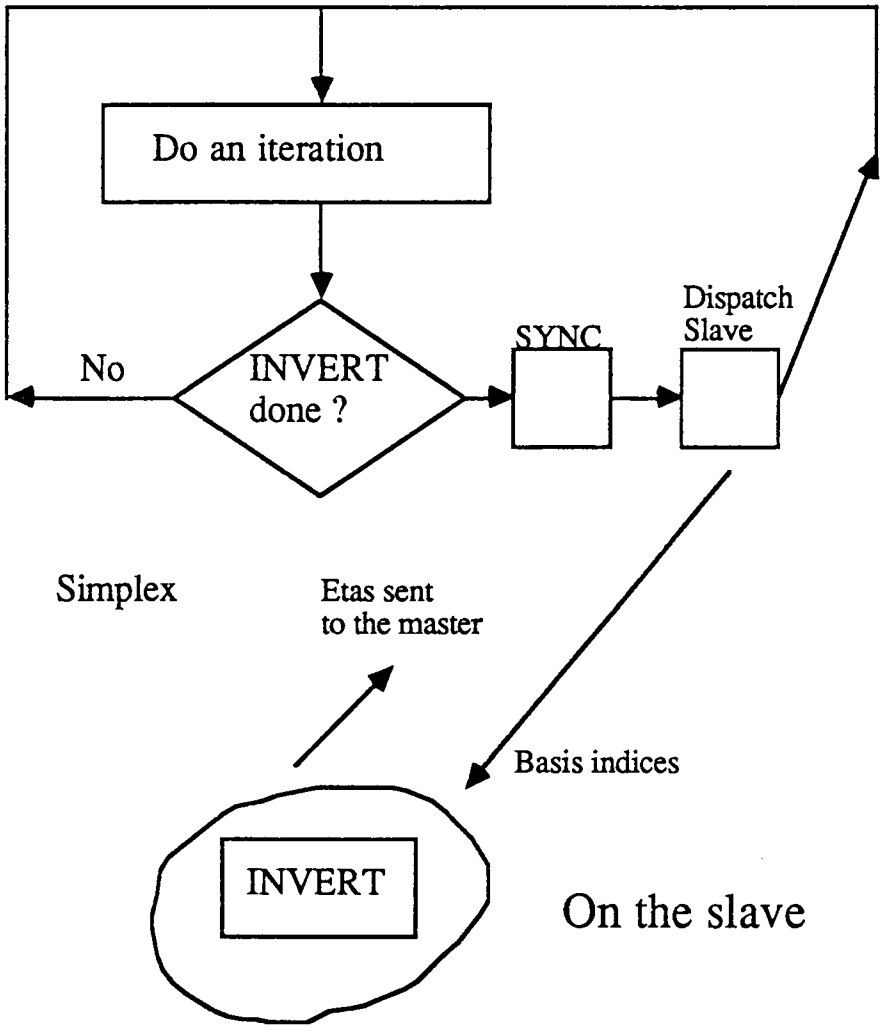
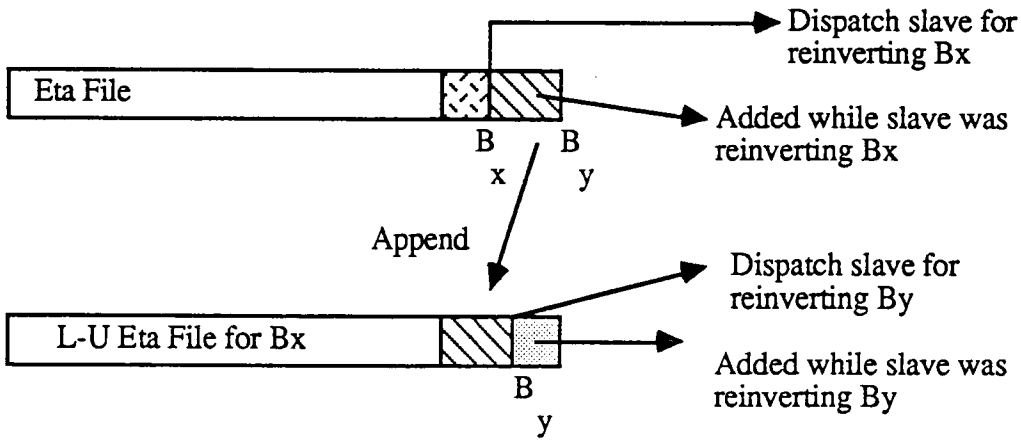


Figure 5.3 : Flowchart of the DINV Algorithm



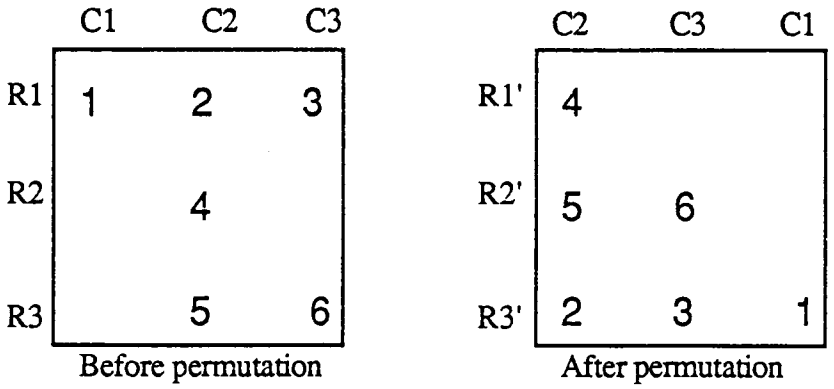
- All shaded regions indicate PFI part of the eta file
- Unshaded regions indicate EFI part of the eta file

Figure 5.4 : Eta file profile at consecutive slave dispatches

(a) During reinversion, the slave permutes the basis matrix by a suitable permutationmatrix (say $\mathbf{P}_i$) for the purpose of generating a sparse eta file (cf. §3.2) and this permutation alters the row indices. Meanwhile, the PFI etas generated by the master are stored (in sparse form; cf. §3.2) using the *unaltered* row indices. Therefore, these etas must also be permuted by $\mathbf{P}_i$ (step 2 of SYNC of Algorithm 5.1). This synchronization step is illustrated by figure 5.5. Figure 5.5a shows an example basis before and after permutation. Figures 5.5b shows the effect of this permutation on the row indices and figure 5.5c illustrates the effect of the permutation on the PFI eta file.
(b) In the second step, the PFI etas generated by the master are appended to the eta file generated by the slave (step 3 of SYNC of Algorithm 5.1).
(c) Finally, the solution vector is updated by the permuted PFI eta file (step 4 of SYNC of Algorithm 5.1).

Performing the reinversions in parallel with the simplex subroutines (FORMC, BTRAN etc. cf. §3.2) implies that reinversion time does not add to the time to solve a problem. That is, one saves completely on the reinversion time. Also, it is seen that the slave is kept busy with the task of reinverting. Due to the continuous operation of the slave, the eta file is sparse and hence, the time to execute the revised simplex subroutines, BTRAN and FTRAN is reduced.

The detailed description of the DINV algorithm requires the following notations. Let $\mathbf{B}_i$ be the basis at the ith simplex iteration and let the corresponding permutation matrix used by INVERT be $\mathbf{P}_i$. Let $\mathbf{x}_i$ be the solution vector at the ith iteration and $\mathbf{F}_i$ represent the L-U factors of $\mathbf{P}_i\mathbf{B}_i$. Let DINV($\mathbf{B}_i$, $\mathbf{P}_i$, $\mathbf{x}_i$, $\mathbf{F}_i$) represent the process of reinverting $\mathbf{B}_i$ on the slave processor. Algorithm 5.1 describes the DINV algorithm.



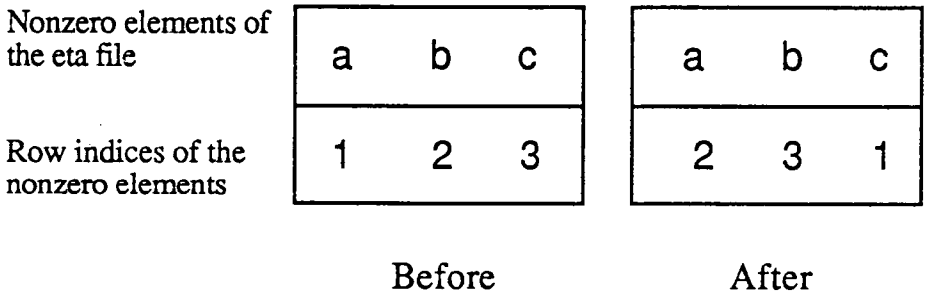
(a) an example basis

$$R1' = R2$$

$$R2' = R3$$

$$R3' = R1$$

(b) : Effect of permutation on the row indices



(c) Effect of permutation on the eta file

Figure 5.5 : Effect of basis permutation on the DINV algorithm

Algorithm 5.1 :

/* On the master processor */

1. Complete one iteration, dispatch DINV($\mathbf{B}_1$, $\mathbf{P}_1$, $\mathbf{x}_1$, $\mathbf{F}_1$) and set i* to 1.

/* i* represents the iteration number when the latest DINV(*,*,*,*) was dispatched */

2. For each subsequent iteration i

   a. Do an iteration and if optimization is complete goto 3. Else proceed.

   b. Append the update of the entering column, $\mathbf{E}_i$, to the eta file.

   c.

   Case c1 (Eta file is full)

   - Set i* to i, dispatch DINV($\mathbf{B}_i$, $\mathbf{P}_i$, $\mathbf{x}_i$, $\mathbf{F}_i$) and wait until it is done.

   Case c2 (Otherwise)

   - If DINV($\mathbf{B}_{i*}$, $\mathbf{P}_{i*}$, $\mathbf{x}_{i*}$, $\mathbf{F}_{i*}$) is done then

     call SYNC(i*, $\mathbf{P}_{i*}$, $\mathbf{x}_{i*}$, $\mathbf{F}_{i*}$, i, $\mathbf{x}_i$) and

     dispatch DINV($\mathbf{B}_i$, $\mathbf{P}_i$, $\mathbf{x}_i$, $\mathbf{F}_i$).

EndFor

3. Terminate all processes.

SYNC(i*, $\mathbf{P}_{i*}$, $\mathbf{x}_{i*}$, $\mathbf{F}_{i*}$, i, $\mathbf{x}_i$) :

1. Incorporate $\mathbf{F}_{i*}$ into the eta file at the master processor.

2. Permute each PFI eta matrix $\mathbf{E}_k$ (i*+1 $\leq$ k $\leq$ i) using the permutation matrix $\mathbf{P}_{i*}$.

3. Append the permuted PFI eta matrices to the etas of $\mathbf{F}_{i*}$.

See figure 5.4.

4. Compute $\mathbf{x}_i = \mathbf{E}_i^{-1} \ldots \mathbf{E}_{i*+1}^{-1} \mathbf{P}_{i*}^{-1} \mathbf{x}_{i*}$

5. Set i* to i and RETURN

DINV($\mathbf{B}_j$, $\mathbf{P}_j$, $\mathbf{x}_j$, $\mathbf{F}_j$) /* On the slave processor */

1. Wait to be dispatched by the master.

2. INVERT($\mathbf{B}_j$, $\mathbf{P}_j$, $\mathbf{x}_j$, $\mathbf{F}_j$)

3. Signal completion of the reinversion to the master.

4. Goto 1

INVERT($\mathbf{B}_j$, $\mathbf{P}_j$, $\mathbf{x}_j$, $\mathbf{F}_j$) :

1. Determine $\mathbf{P}_j$.

2. Generate $\mathbf{F}_j$, the L-U factors for $\mathbf{P}_j\mathbf{B}_j$.

3. Compute $\mathbf{x}_j = \mathbf{B}_j^{-1}\mathbf{P}_j^{-1}\mathbf{b}$ where $\mathbf{b}$ is the right hand side.

4. RETURN

5.2.2 Implementation of Distributed INVERT

In the above pseudo-code, the underlined steps require interprocessor communication. Interprocessor communication for the DINV algorithm is required when the master must *dispatch* the slave to reinvert a given basis and when the slave must *signal* to the master that it is done with reinverting. In this section we describe how these issues are

dealt with for the two types of MIMD computers, the distributed memory and shared memory computers.

Algorithm 5.2 describes the implementation of the DINV algorithm on the hypercube iPSC/2 (discussed in §2.4.1), an example of distributed memory MIMD computers. The implementation on the iPSC uses a host processor in addition to the master and slave processors and these processors run the programs HST, SX and DI respectively. The host processor serves as an administrative console and as a "gateway" to the hypercube for users. To run the DINV algorithm, the user logs on to the host, acquires a one-dimensional cube (two processors) by issuing the iPSC command, "getcube" to the host, and loads the HST program on the host. The HST program reads the problem's data into DATA_H, loads programs SX and DI on the master and slave processors, sends the data to the processors and kills all the processes when the algorithm is completed. The problem's data is received by the master into DATA_S and by the slave into DATA_I.

Communication between the master and the slave is achieved by passing messages. The master dispatches the slave by sending the basis indices (message) stored in BI_S. The slave, which has been waiting for this message, receives it in BI_P and proceeds to reinvert the basis. Simultaneously, the master executes a simplex iteration and then checks to see if the slave is done with reinverting. This involves checking the system buffer for a message (the eta file) from the slave. If there is none, the master proceeds to execute simplex iterations until the slave is done with its task. Due to the execution of simplex iterations, the solution vector and the basis indices change and PFI etas are added to the eta file. X, TMPBAS and PFI, respectively, are used to save these changes. Once the reinversion is done, the slave signals to the master by sending a message (called eta file-message) consisting of : the eta file

generated by INVERT (stored in EFI_I); the permutation sequence (stored in PERM_I) applied to invert the given basis; and the solution vector (stored in X_I) computed using this eta file. The master receives this message into EFI_S, PERM_S, and X_S, performs the synchronization tasks (SYNC of Algorithm 5.1) and then dispatches the slave again. These programming issues, discussed above, have been summarized in Tables 5.2 and 5.3.

Algorithm 5.2 : /* Distributed memory DINV implementation */

Host /* On the host processor */

Storage : DATA_H

1. Load programs on the nodes.

2. Read data into DATA_H

3. CSEND DATA_H

4. Wait for termination

5. Kill all processes and STOP

SX /* On the master processor */

Storage : DATA_S, PFI, TMPBAS, X, EFI_S, X_S, PERM_S, BI_S

1. CRECV in DATA_S

2. CRECV initial etas in EFI_S

3. Complete an iteration and if optimized goto (8) (cf. NORMAL in §3.2)

4. IPROBE for message from slave and if none goto (3). Else proceed.

5. CRECV the eta file message into EFI_S, X_S and PERM_S :

That is, the master copies the message from its system buffer to its storage area.

Note that the time spent by the master processor waiting for the eta file is zero.

Table 5.2 : Storage requirements for the DINV Algorithm

Storage for	iPSC		Sequent Balance	
	Master	Slave	Master	Slave
Data	Yes	Yes	Yes	No
EFI etas	Yes	Yes	Yes	Yes
PFI etas	Yes	No	Yes	No

Table 5.3 : Communication mechanisms for the DINV algorithm

Type	iPSC	Sequent Balance
Waiting to be dispatched by master	CRECV	M_PARK_PROCS
Dispatch slave	CSEND basis indices	M_RELE_PROCS
Checking for slave completion	IPROBE	By memory variables
Incorporating EFI etas (on master)	CRECV	Copying
Signalling completion to master	CSEND etas	By memory variables
Termination by master	CSEND message to kill	M_KILL

6. Do steps 2 thru 4 of SYNC (cf. Algorithm 5.1)

7. CSEND basis indices message (BI_S) to slave and goto (3) :

a. Copy BI_S from master's program area to its system buffer.

b. Send system buffer across the communication channel.

8. CSEND termination message to *Host.*

DI

Storage : DATA_I, EFI_I, X_I, PERM_I, BI_I

1. CRECV in DATA_I

2. Do initial INVERT

3. CSEND eta file message (EFI_I, X_I and PERM_I) to master :

a. Copy the information from program's area to system buffer.

b. Send system buffer across communication channel.

4. Wait to receive basis indices message into BI_I; that is, call CRECV.

a. Wait for the basis indices-message

b. Copy message from system buffer to program's area.

5. DO INVERT

6. Goto (4)

From Algorithm 5.2, we see that the problem's data is stored on all the three processors, the EFI etas are stored on the master and slave processors and the PFI etas are stored only on the master. Let n_{data} bytes of storage be allocated for the problem's data, n_{efi} bytes be allocated for the EFI etas and n_{pfi} bytes be allocated for the PFI etas. Since the storage requirements for X_x, PERM, BI_x and TMPBAS is not significant, the total amount of memory for implementing the DINV algorithm in a distributed memory computing environment is approximately

$3n_{data} + 2n_{efi} + n_{pfi}$.

We now consider the implementation of the DINV algorithm on the Sequent Balance (cf. §2.4.2), an example of a shared memory MIMD computer. This implementation, presented below in Algorithm 5.3, uses only two processors, called master and slave, running the programs SX and DI respectively. The program SX initiates the algorithm, executes the simplex iterations, creates and dispatches the slave processor anf finally kills all the processes at optimality. Since the problem's data is never altered during the solution process, it is stored in shared memory (DATA).

Interprocessor communication is required when the master must dispatch the slave to reinvert a given basis and when the slave must signal to master that the reinverion is completed. This communication is achieved through the use of a flag, called NFINIS, allocated in shared storage. NFINIS is set to zero by the master to dispatch the slave. While the slave is reinverting, the master processor, using the eta file generated by the previous slave dispatch, continues to execute simplex iterations until the slave is done with reinverting. This is indicated when the slave sets NFINIS to one. At this point, the eta file generated by INVERT, the permutation sequence applied to reinvert the basis, and the solution vector reside in shared memory in EFI, PERM and X respectively. The master copies this information into its private storage area (EFI_P, PERM_P and X_P), sets NFINIS to zero to dispatch the slave and the same process repeats. See Tables 5.2 and 5.3.

Algorithm 5.3 : /* Shared memory DINV implementation */

SX /* On the master processor */

Shared storage : DATA, EFI, X, PERM, BI, NFINIS

Private storage : X_P, PFI, TMPBAS, EFI_P, BI_P

1. Read data into DATA.

2. Do initial INVERT and one simplex iteration.

3. Copy BI_P to BI

4. a. Call M_SET_PROCS(2)

   b. Call M_FORK(*DI*)

5. Complete an iteration and if optimized goto (8) (cf. NORMAL in §3.2)

6. Check for completion of reinversion and proceed "appropriately :"

   a. Call M_LOCK

   b. If NFINIS is zero then the slave is busy. Hence, goto (6i)

   c. Call M_PARK_PROCS

   d. Copy EFI to EFI_P

   e. Do steps 2-4 of SYNC (cf. Algorithm 5.1)

   f. Copy BI_P to BI

   g. Set NFINIS to 0

   h. Call M_RELE_PROCS

   i. Call M_UNLOCK

7. Goto (5)

8. Call M_KILL_PROCS and STOP

DI

Shared storage : DATA, EFI, X, PERM, BI, NFINIS

1. Get MYID. If MYID="master" then RETURN. Else proceed.

2. Do INVERT

3. Signal completion :

   a. Call M_LOCK

   b. Set NFINIS to 1

   c. Call M_UNLOCK

4. RETURN

From Algorithm 5.3, we see that the problem's data is stored in shared memory, the EFI etas are stored in shared memory and in private memory of the master, and PFI etas are stored in private memory of the master. Since the storage requirement for X, PERM_S and BI_x is not significant, the total amount of memory required for the DINV algorithm on a shared memory computer is approximately

$$n_{data} + 2n_{efi} + n_{pfi}.$$

5.2.3 Time Taken by Distributed INVERT

In chapter 4, timing models were developed for the revised simplex algorithm and its modules and those models were used to infer the time taken by the algorithm. In this section, the timing model for the revised simplex method is used in the prediction of the run-time of the DINV algorithm. More specifically (in §5.2.3.3), we use the timing model to predict the time for the simplex iterations of the DINV algorithm.

Between consecutive dispatches of the slave, the master processor executes : (a) simplex iterations; (b) tasks required to support the interprocessor communication required for dispatching the slave and for receiving the eta file from the slave and (c) synchronization tasks (SYNC of Algorithm 5.1) when the slave is done. We define a DINV cycle to consist of the tasks executed on the master between two consecutive dispatches of the slave.

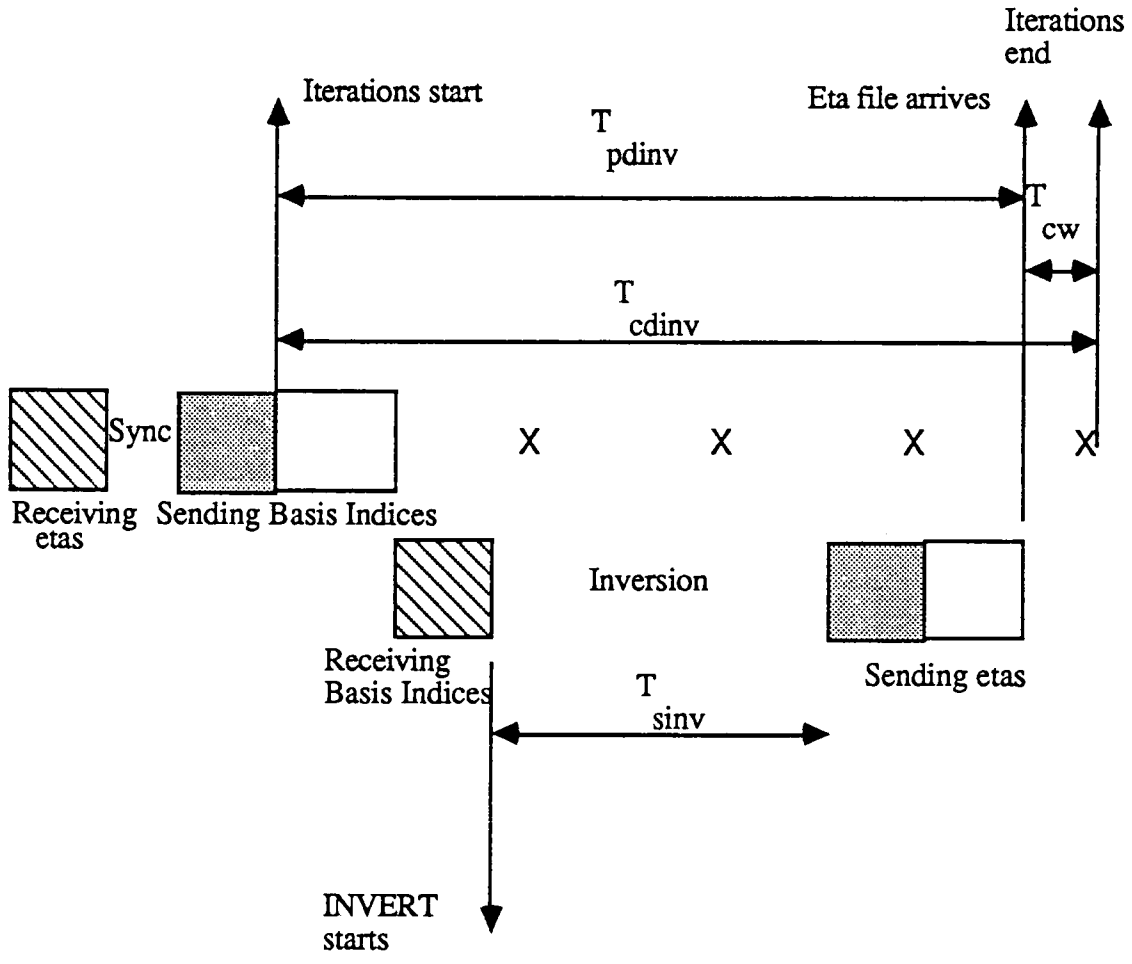
Since the master processor starts and ends the DINV algorithm, the time taken by the DINV algorithm to solve a problem is the time for which the master is busy. This

can be computed if one knows the number of DINV cycles executed by the master and the time per DINV cycle.

5.2.3.1 Time Taken by Distributed INVERT on Distributed Memory Computers

In a distributed memory computing environment, the interprocessor communication overhead required by the algorithm is the time spent by the master processor for receiving the eta file-message (EFI_I, X_I, PERM_I of Algorithm 5.2) from the slave and sending the basis indices-message (BI_S of Algorithm 5.2) to the slave (see figure 5.6). Reference to §5.1 shows that the time spent by the master to support these communications is the time for transferring (read from or write to) the message between program's storage area and the system buffer. Thus, the communication overhead is the time for steps 5 and 7a in the program SX of Algorithm 5.2. Let t_{cop} be the time to transfer a one byte message between the system buffer and the program's storage area. Let n_{bi} be the number of bytes allocated to store the basis indices and n_{efi} be the number of bytes allocated to store the EFI eta file (eta file generated by INVERT). Then, the communication overhead time per DINV cycle is $t_{cop}*(n_{bi} + n_{efi})$.

The time to execute the simplex iterations in a DINV cycle is termed cycle iterations time and is denoted by T_{cdinv}. Once the basis indices are copied from the storage area of the master processor to its system buffer, the master executes simplex iterations until it detects the presence of an eta file message in its system buffer (see figure 5.6). Hence, the components of T_{cdinv} are : (a) the time to relay the basis indices message across the communication link to the slave processor; (b) the time



X Indicates end of an iteration

Copying from system buffer to program's storage area

Copying to system buffer from program's storage area

Relay message across channel

Figure 5.6 : DINV process on distributed memory computers

spent by the slave processor to receive the basis indices; (c) the time for executing INVERT (denoted by T_{sinv}); (d) the time to send the eta file from the slave processor's storage area to the master processor's system buffer and (e) the amount of time (denoted by T_{cw}) the eta file waits at the system buffer of the master processor. Let t_{chnl} be the time to relay a one byte message between the system buffers of the master and slave processors. The cycle iterations time (T_{cdinv}) is therefore, $t_{chnl}*n_{bi} + T_{sinv} + (t_{cop} + t_{chnl})*n_{efi} + T_{cw}$.

The synchronization time per DINV cycle is the time for step 6 in the SX program on Algorithm 5.2 (or steps 2 thru 4 of SYNC in Algorithm 5.1) and is denoted by T_{aoh}. An approximate inference of T_{aoh} is as follows. The main step is step 4 of SYNC of Algorithm 5.1 and this involves an FTRAN. Let T_{SFTRAN} be the total serial time for executing K FTRAN's where K is the number of iterations to solve the problem. Since K/C_{itr} FTRAN's are performed, T_{aoh} is T_{SFTRAN} / C_{itr}.

Let K be the number of iterations to solve the problem and C_{itr} be the average number of iterations in a DINV cycle. Then, the number of DINV cycles is K/C_{itr} and hence the time for the DINV algorithm on distributed memory computers is given by (5.3).

$$T_{DINV} = (K/C_{itr}) * [\, t_{cop}*(n_{bi} + n_{et}) + T_{cdinv} + T_{aoh} \,] \qquad (5.3)$$

Using (5.3), T_{DINV} can be computed provided the parameter C_{itr} is known. In §5.2.3.3, we provide a method for inferring it.

5.2.3.2 Time Taken by Distributed INVERT on Shared Memory Computers

The interprocessor communication time per DINV cycle is the time for executing the calls to M_LOCK-M_UNLOCK, M_PARK_PROCS and M_RELE_PROC (cf. §2.4.2) and they are steps 6a, 6c, 6h and 6i in the SX program of Algorithm 5.3. The cycle iterations time (T_{cdinv}) has the following components : (a) the serial inversion time T_{sinv}; (b) the wait time T_{cw} and (c) the time for steps 3a and 3c in the DI program of Algorithm 5.3. T_{cdinv} is therefore, $t_{lck} + T_{sinv} + T_{cw}$. The synchronization time per DINV cycle (T_{aoh}) can be inferred using the method proposed in §5.2.3.1.

In addition to the terms/notations defined in the previous section we introduce the following. Let t_{lck}, t_{prk} and t_{rel} be, respectively, the times to execute the calls to the M_LOCK-M_UNLOCK construct, the M_PARK_PROCS routine and the M_RELE_PROCS routine. The number of DINV cycles is K/C_{itr} and hence the time for the DINV algorithm on a shared memory computer is given by (5.4)

$$T_{DINV} = (K/C_{itr}) * [(t_{lck}+t_{prk}+t_{rel}) + T_{cdinv} + T_{aoh}] \qquad (5.4)$$

Using (5.4), T_{DINV} can be computed provided the parameter C_{itr} is known. In §5.2.3.3, we provide a method for inferring it.

5.2.3.3 Number of Simplex Iterations in a Distributed INVERT Cycle

The number of iterations executed in a DINV cycle depends on the elapsed time (T_{pdinv}) between the start of the iterations after a slave dispatch and the arrival of the corresponding eta file at the master. Note that T_{pdinv} is actually the cycle iteration time minus the wait time T_{cw}. The components of T_{cdinv} have been identified in the last two sections and from there the value of T_{pdinv} can be inferred. The question addressed in this section is : "Given the value of T_{pdinv}, how many simplex iterations can be executed in x DINV cycles?"

At the end of each simplex iteration, the master checks for the arrival of the eta file (checks discussed in §5.2.2). If the eta file has arrived, then the master executes synchronization steps (SYNC of Algorithm 5.1) and initiates the next DINV cycle by dispatching the slave. Otherwise, the master executes one more iteration in the current DINV cycle. For example, suppose T_{pdinv} is 100 time units and each simplex iteration takes the same amount of, say, 35 time units. Then, the eta file would arrive while the master is executing the fourth iteration. In other words, C_{itr} is 4 and it is given by Ceil(T_{pdinv} / 35) where Ceil(x) gives the smallest integer larger than or equal to x. This method, however, will not be correct if the simplex iterations in a DINV cycle do not take the same amount of time.

Reference to §3.2 shows that during a DINV cycle, the eta factorization of the basis is maintained using the PFI (product form of inverse) scheme and in that scheme a PFI eta is added to the eta file at the end of each simplex iteration. The growth of the eta file implies that the time for the revised simplex modules, BTRAN and FTRAN,

increases and this increase causes the time for simplex iterations to increase. This increase, as we saw in §4.2, can be described by a *linear growth model.* If t_0 and t_1 are, respectively, the intercept and slope of the linear growth model, then the addition of a PFI eta causes the iteration time to increase by t_1 time units. That is, given that we start a DINV cycle with j PFI etas, the time taken by the master processor to execute the first iteration of that cycle is $t_0 + (j+1)*t_1$ time units and the time to execute the first k iterations is :

$$T(j,k) = \sum_{i=j+1}^{j+k} (t_0 + i*t_1) \qquad (5.5)$$

Our method to infer the number of simplex iterations is based on this fact. The following examples illustrate the method. For ease of explanation, we consider the implementation on distributed memory computers.

Example 1 : Let $t_0 = 32$, $t_1 = 3$, $T_{pdinv} = 90$ and $K = 1000$.

The first DINV dispatch is started after one iteration (cf. §5.2.1). Hence, at the start of the first DINV cycle the eta file on the master processor contains one PFI eta. By eqn (5.5), the first iteration in the first DINV cycle takes 38 time units [= 32 + (1+1)*3] and since T_{pdinv} is 90, the eta file would still not have arrived at the master's system buffer. The time for the first two iterations of the first DINV cycle is 79 (= 38 + 41) and the time for the first three iterations is 123 (= 79 + 44). Therefore, the eta file would arrive while the master is executing the third iteration and would be waiting

for 33 time units (= 123-90) at the master's system buffer. The second cycle is started with three PFI etas in the eta file. By eqn (5.5), the time to execute the first iteration in the second cycle is 44 [= 32 + (3+1)*3], the time for the first two iterations of the second cycle is 91 (= 44 + 47) and hence the master completes two iterations during this cycle. During the third cycle, which is started with two PFI etas, the time to execute the first iteration is 41 [= 32 + (2+1)*3], the time to execute the first two iterations is 85 (= 41 + 44) and the time to execute the first three iterations is 132 (= 85 + 47). Hence, three iterations are executed during the third cycle; that is, the fourth cycle is started with three PFI etas.

Since the second cycle was also started with three PFI etas, the number of iterations executed in the fourth and the second cycles must be the same (=2). This in turn implies that the fifth and the third cycles will start with two PFI etas and will therefore have the same number of iterations (= 3). In summary, the number of iterations in the ith and the (i+2)nd cycles will be the same ($i \geq 2$). Thus, after the first DINV cycle, five iterations are executed in two consecutive DINV cycles ($C_{itr} = 5$) and by the growth model the time for those five iterations is 223 ($T_{cdinv} = 91 + 132$). Hence, the time for executing all the simplex iterations of the DINV algorithm is (1000/5)*223 [= $(K/C_{itr})*T_{cdinv}$].

Example 2 : Let $t_0 = 25$; $t_1 = 10$; $T_{pdinv} = 90$; $K = 1000$.

The first DINV cycle is started with one PFI eta and execute two iterations in that cycle. During the second DINV cycle, which is started with two PFI etas, two iterations are executed (time to execute two iterations is 110) and hence will start the third cycle with two PFI etas.

Since the second cycle was also started with two PFI etas, the number of

iterations in the third and second cycles must be the same. This inturn implies that the fourth cycle is a replication of the third (and second) and so on. Thus, after the first DINV cycle, two iterations are executed in every DINV cycle ($C_{itr} = 2$) and the time for those two iterations is 110 time units ($T_{cdinv} = 110$). Hence, the time for executing all the simplex iterations of the DINV algorithm is (1000/2)*110 [$=(K/C_{itr})*T_{cdinv}$].

Let CITR(i) represent the number of iterations in the ith DINV cycle and let the sequence {CITR(1), CITR(2),...} be termed the eta sequence (ES). The examples above illustrate how the time for the DINV algorithm can be inferred. In example 1, CITR(j) = CITR(j+2) [j≥2] and in example 2, CITR(j) = CITR(j+1) [j≥2]. That is, in both the examples, the number of iterations in a certain cycle was the same as that in a previous cycle and this served as the basis for the inference of the algorithm's time. Stated mathematically, *suppose* there exists j* and t such that CITR(j) = CITR(j+t) for all $j \geq j^*$, then the number of iterations in t DINV cycles, the time for those t cycles and the time for the simplex iterations of the DINV algorithm are, respectively, given by :

$$\sum_{j=j^*}^{j+t} \text{CITR}(j) \qquad (5.6)$$

$$\sum_{j=j^*}^{j+t} T(\text{CITR}(j-1), \text{CITR}(j)) \qquad (5.7)$$

[(Result of (5.7)) / (Result of (5.6))] * K (5.8)

where T(*,*) is computed using eqn (5.5).

We shall now show that the above supposition is valid if the number of DINV cycles is ≥ CITR(1). To show this, it suffices to show that one of the elements in the ES reappears.

The first cycle is started with the minimum number of etas (=1) and hence, the maximum number of iterations must be executed during this cycle. That is, the number of iterations in a DINV cycle must be ≤ CITR(1). This implies that the number of unique elements in the ES is ≤ CITR(1). Since the number of elements in the ES [≥ CITR(1)] is larger than the number of unique elements, at some point an element must reappear.

5.2.4 Results with Distributed INVERT Algorithm

The DINV algorithm was implemented on the iPSC/2, a distributed memory MIMD computer. For our test problems, the serial time using an inversion frequency of 50, and the DINV algorithm's run time appear, respectively, in columns 3 and 4 of Table 5.4. To infer the algorithm's time from eqn (5.3), we assume that the time to transfer a byte from the program's area to the system buffer to be the same as the amount of time to transfer a byte between two locations in the program's area. These inferences appear in column 5. For the four starred problems of Table 5.4, the DINV algorithm did not provide a siginificant saving and for two of those cases (SHIP04S and SHIP04L) the DINV algorithm took longer to solve the problem. In these four cases the PFI eta density is about 8 as compared to an average of nearly 63 for our test problems. The low PFI eta density implies that the reduction in BTRAN and FTRAN times is not significant and therefore, the savings due to the DINV algorithm is not considerable in these cases. The weighted saving (savings = serial time - DINV time) and inferred weighted saving with respect to the serial run time is 21.66% and 23.73% for our test problems. Surprisingly, the message time is not substantial. The weighted message time is only 1.4% of the serial run time. We therefore conclude that the potential

Table 5.4 : Results with the DINV algorithm

Problem	# of rows	Serial run time in msecs	Actual time of DINV algorithm	Inferred time of DINV algorithm	Message time for DINV algorithm
agg	489	16697	14767	14588	7.9
agg2	517	22708	21035	22110	8.6
agg3	517	24071	21876	22948	8.3
bandm	306	189401	148688	148210	2.3
brandy	221	88586	69435	68879	4.2
israel	175	53108	39444	37459	5.5
sc205	206	24303	19366	18365	5.4
scagr25	472	159537	115176	124311	1.7
scfxm1	331	65999	55064	53055	3.3
scfxm2	661	283245	214952	208982	0.9
scfxm3	991	736687	482551	478716	0.3
scorpion	389	39969	31532	32174	3.0
scrs8	491	203560	161431	158708	1.5
scsd6	148	66382	63868	60533	7.0
scsd8	398	399428	345612	322468	1.4
sctap1	301	50461	42601	43189	5.6
sctap2	1091	526241	409933	403181	0.8
sctap3	1481	1126754	775867	772125	0.3
ship04l*	403	145231	147502	142156	6.2
ship04s*	403	73887	74804	71358	6.4
ship08s*	779	195716	177875	169470	2.0
ship12s*	1152	462290	384214	367233	0.9
zdeep1	101	55578	47489	48073	3.4
zdeep2	101	40234	34410	34363	3.5
zdynamico	678	1020908	829867	784927	0.5
zegd051	869	127708	106035	103782	2.2
zegd085	1443	398422	295522	288227	0.8
zfixmar	325	62072	52257	50658	3.4
zforestry	402	197253	152455	153201	1.2
zmeat12b	381	81004	66238	64598	2.8
zmeat31b	384	88077	71412	68753	1.9
zmeat43b	648	140301	100785	98772	1.0
zmege06	1011	98568	93105	88138	2.9
zmege08	1343	200714	174962	166141	1.2
zmrp3	301	35847	34048	31461	6.3
	Mean weighted saving		21.66	23.73	1.41
	Mean saving		16.83	18.55	

improvement in the DINV algorithm due to the decrease in message time is limited to about 1.4%.

5.3 Distributed PRICE

5.3.1 Distributed PRICE Algorithm

PRICE is the revised simplex module for determining the entering column. The Distributed PRICE (DPRI) algorithm uses multiple (say 0 thru N) processors to determine the entering column at each simplex iteration. The algorithm consists of partitioning the set of nonbasic columns into N+1 disjoint sets, NB_0, ..., NB_N and distributing them among the N+1 processors as follows. Processor 0, called the master processor, executes the simplex iterations. When the entering column is to be found, it dispatches processors 1 thru N with the task of, respectively, pricing the columns in NB_1, ..., NB_N and then proceeds to price the columns in NB_0. Processors 1 thru N (called slaves) complete their assigned share of pricing and report to the master : (a) DJ_p, the minimum reduced cost among the columns in NB_p and (b) J_p, the corresponding column index.

Given the dual vector, π let DPRI(π, NB_p, DJ_p, J_p) ($1 \le p \le N$) represent the process which prices the columns in NB_p and returns DJ_p and J_p. The DPRI algorithm can now be described as follows.

Algorithm 5.4 :

DPRI : P_{SX}

A. For each iteration i

1. Perform FORMC and BTRAN

2. Dispatch DPRI(π, NB_p, DJ_p, J_p) for p = 1, ... ,N

3. PRICE(π, NB_0, DJ_0, J_0)

4. Set Reduced cost = DJ_0 and set entering column = J_0 if $DJ_0 = 0$

5. Determine entering column and if none goto (B)

For p = 1,N

a. Wait for DPRI(π, NB_p, DJ_p, J_p) to finish

b. If (DJ_p < Min Reduced cost) then

set Min Reduced cost = DJ_p , Entering column = J_p

Endfor

6. Remainder of SIMPLEX (cf. §3.2)

Endfor

B. Terminate.

DPRI : DPRI(π, NB_p, DJ_p, J_p)

1. Wait to be dispatched

2. PRICE (π_p, NB_p, DJ_p, J_p)

3. Signal completion and goto (1).

PRICE(π, NB_p, DJ_p, J_p)

Set DJ_p and J_p.

5.3.2 Implementation of Distributed PRICE

In the above pseudo-code, the underlined steps require interprocessor communication. Algorithms 5.5 and 5.6 elaborate on how communication is achieved in distributed and shared memory implementations of the DPRI algorithm. Another issue in the DPRI implementation is the method used to partition the columns among the processors. Suppose we use two processors, one of which prices all the sparse columns and the other prices all the dense columns. In this case, the workload is not evenly distributed among the processors and effectiveness of the algorithm is therefore reduced. Therefore, the efficiency of the algorithm depends on knowing the indices of the sparse and dense columns and then distributing them appropriately among the processors. In our data structure, logical columns are stored first and then the structural columns are stored. We let the pth processor price the pth, the (p+N)th, the (p+2N)th columns and so on. Thereby, the logical and structural columns are divided equally among the processors. Other partitions are possible. For instance, the first processor can be assigned the first n/N columns (n is the number of nonbasic columns), the second processor can be assigned the second n/N columns and so on.

Algorithm 5.5 : /* For distributed memory computers */

Host

/* On the host processor. Same as the HOST program of the DINV algorithm. */

S X

Storage : DATA_S, PI_S, BI_S, DJ_0, J_0, + other RSM storage

/*

DATA_x is for storing the problem's data. PI_x contains the dual vector. BI_x is for storing the basis indices. DJ_p contains the minimum reduced cost among the columns in NB_p. J_p contains the corresponding column index.

*/

1. CRECV data in DATA_S
2. For each iteration
    a. ISEND BI_S to all processors.
    b. Do FORMC and BTRAN
    c. CSEND PI_S to all processors.
    d. PRICE(DATA_S, PI_S, BI_S, DJ_0, J_0)
    e. Determine the entering column as follows :
        For p = 1, N
            - CRECV DJ_p and J_p
            - IF (DJ_p ≤ DJ_0) DJ_0 = DJ_p and J_0 = J_p
        EndFor
    f. Do FTRAN, CHUZR, WRETA, UPDATE

DPRI /* On the slave processors */

Storage : DATA_p, PI_p, BI_p, DJ_p, J_p

1. CRECV data into DATA_p
2. CRECV basis indices into BI_p
3. CRECV dual vector into PI_p
4. PRICE(DATA_p, PI_p, BI_p, DJ_p, J_p)

5. CSEND DJ_p and J_p

6. Goto 2

Subroutine PRICE(DATA_p, PI_p, BI_p, DJ_p, J_p)

p <---- ID of the processor (master is 0 and slaves are numbered 1 thru N)

For i starting from p+1 to number of columns in steps on N+1

- Price column i

- If (reduced cost of column i $\leq DJ_p$) then update DJ_p and J_p

EndFor

Algorithm 5.6 : /* Shared memory implementation of the DPRI algorithm */

SX /* On the master processor */

Shared Storage : DATA, DJ(0:N), J(0:N), PI, + other RSM storage

1. Read data into DATA

2. Call M_SET_PROCS(N+1)

3. For each iteration i

a. Do FORMC, BTRAN

b. Call M_FORK(PRICE)

c. Call M_PARK_PROCS

d. Decide on the entering column and if none goto(3)

e. Do FTRAN, CHUZR, WRETA, UPDATE

Endfor

4. Call M_KILL and STOP

Subroutine PRICE

Shared Storage : DATA, DJ(0:N), J(0:N), PI, + other RSM storage

1. p <--- ID of the processor (master is 0, slaves are numbered 1 thru N)

2. For i starting from p+1 to number of columns in steps on N+1

- Price column i

- If (reduced cost of column i ≤ DJ(p)) update DJ(p) and J(p)

EndFor

3. Call M_SYNC

4. RETURN

5.3.3 Time Taken by Distributed PRICE

In the DPRI algorithm, the basis indices and the dual vector have to be communicated from the master to the slaves and the minimum reduced cost among the columns in NB_p has to be communicated from the slaves to the master. We consider the DPRI algorithm using one slave processor.

In the distributed memory implementation of the DPRI algorithm, the basis indices are sent by calling ISEND (cf. Algorithm 5.5). The time for this call is negligible, since the master only posts a request to send the message and then continues with the execution of the program (cf. §5.1). To send the dual vector, the master calls CSEND. The message is copied to the system buffer, a link is established and the message is then communicated across the link. As shown in figure 5.7, the master is busy with the sending process only for the time it takes to copy the information from the program's area to the system buffer (cf. §5.1). The communication of the minimum reduced cost from the slave to the master is done along the same lines. See figure 5.7.

Let t_{cop} be the time to copy a one byte message between the system buffer and the program's area, t_{lnk} be the time to establish a link across the communication

Figure 5.7 : DPRI algorithm using two processors

channel, and t_{snd} be the time to send a one byte message across the link. Let n_{dv} be the number of bytes to store the dual vector and n_{dj} be the number of bytes to store the reduced cost. Then, for each simplex iteration, the time to communicate the dual vector from the master's pregram area to the slave's is $t_{lnk} + n_{dv}*(t_{cop}+t_{snd}) + n_{dv}*t_{cop}$ and the time to communicate the reduced cost from the slave to the master is $t_{lnk} + n_{dj}*(t_{cop}+t_{snd}) + n_{dj}*t_{cop}$. Therefore, in a distributed memory computing environment, the time to determine the entering column using the DPRI algorithm is given by (5.9)

$$K * [2*t_{lnk} + (t_{cop}+t_{snd})*(n_{dv}+n_{dj}) + t_{cop}*(n_{dj}+n_{dv})] + T_{SPRICE} / 2 \quad (5.9)$$

where K is the number of iterations and T_{SPRICE} is the serial run time for PRICE.

In a shared memory implementation of the DPRI algorithm, the time for interprocessor communication is the time for steps 3b and 3c of the SX program and step 3 of subroutine PRICE. Let t_{frk}, t_{prk} and t_{sync} be the times to execute the M_FORK, M_PARK_PROCS and M_SYNC calls respectively. Then, the time for determining the entering column using a shared memory version of the DPRI algorithm is given by (5.10).

$$K*[t_{frk}+t_{prk}+t_{sync}] + T_{SPRICE} / 2 \quad (5.10)$$

5.3.4 Results with the distributed PRICE algorithm

The DPRI algorithm was implemented on the iPSC/2, a distributed memory MIMD computer. The combined effect of the DINV and DPRI algorithms on the solution of a problem was studied through a series of runs as follows. In the first run, the test problems were solved using two slave processors, one running INVERT (DINV algorithm) and the other running PRICE on the even nonbasic columns (DPRI algorithm). Note that the master processor is executing simplex iterations and is pricing of the odd nonbasic columns. The savings in the run time to solve a problem using this combined DINV-DPRI algorithm over the run time of the DINV algorithm alone appear in column 3 of Table 5.5. If the run time improved, then the problem was solved again using one more processor for performing the task of pricing. This procedure was continued as long as the run time kept improving. As expected, the run time improvement was in general larger for problems with more columns.

Next, the DPRI time using one slave processor was inferred from eqn (5.9) for our test problems. The inferences pointed to two ways of improving the DPRI algorithm. First, the message time (column 5 of Table 5.6) required for this algorithm is high when compared with that for the DINV algorithm. Hence, an implemenatation of the algorithm on shared memory computers would provide significant improvements in the solution time. Second, in many cases it was observed the load was not evenly balanced between the master and slave. Three prominent cases were SCTAP1, SCTAP2 and SCTAP3. For instance, in the set of nonbasic columns, the ratio of the total number of even to odd structural columns was found to be 3.91:1 for SCTAP1, 7.5:1 for SCTAP2 and 6.5 : 1 for SCTAP3. Suppose one of the processors had L%

Table 5.5 : Results with the DPRI algorithm

Problem	# of structural columns	Savings with 1 slave processor†	Savings with 2 slave processors	Savings with 3 slave processors	Savings with 4 slave processors	Savings with DINV-DPRI algorithms
agg	163	-0.35	^			11.6
agg2	302	7.97	7.93			14.8
agg3	302	8.34	7.51			16.7
bandm	472	-0.23	2.58			23.0
brandy	249	-1.91				21.6
israel	142	-3.87				25.7
sc205	203	-9.77				20.3
scagr25	500	-2.33				27.8
scfxm1	457	2.51	0.73			18.7
scfxm2	914	8.90	7.14			30.9
scfxm3	1371	10.53	6.50			41.4
scorpion	358	-5.66				21.1
scrs8	1169	7.72	8.49	5.88		27.4
scsd6	1350	19.34	14.28			22.4
scsd8	2750	21.42	21.37			32.0
sctap1	480	-6.08				15.6
sctap2	1880	6.93	16.80	13.31		35.2
sctap3	2480	7.10	14.86	9.25		41.4
ship04l	2118	28.23	34.88	34.92	33.69	33.9
ship04s	1458	19.64	23.43	21.43		22.5
ship08s	2387	24.68	31.14	32.51	31.39	38.7
ship12s	2763	22.59	28.54	29.94	29.34	41.8
zdeep1	164	4.22	1.73			18.2
zdeep2	125	-0.84				14.5
zdynamicc	498	4.70	6.22			23.8
zegd051	408	-1.86				17.0
zegd085	680	1.58	0.64			27.0
zfixmar	452	3.16	-1.01			18.5
zforestry	603	8.02	-1.82			28.9
zmeat12b	311	-0.45				18.2
zmeat31b	577	8.96	2.63			26.2
zmeat43b	605	9.22	9.36			34.9
zmege06	687	14.02	17.36			21.9
zmege08	908	14.93	17.92	18.45	17.07	28.9
zmrp3	522	-4.48				5.0

Mean weighted savings of DPRI-DINV algo 31.53

† Savings are computed with respect to the serial run time
^ Blank entries indicate that further runs were not made

Table 5.6 : Inference of run time of the DPRI algorithm

Problem	# of structural columns	Serial PRICE time in msecs	Actual pricing time with DPRI algorithm	Inferred pricing time with DPRI algorithm	Message time as a % of actual DPRI time
agg	163	3417	3469	2926	7.1
agg2	302	7919	6242	5605	5.9
agg3	302	8036	6211	5700	5.9
bandm	472	21940	22286	20516	5.6
brandy	249	13342	14669	12382	7.3
israel	142	5886	7414	6017	7.0
sc205	203	1570	3463	2738	9.7
scagr25	500	13294	15979	13457	5.3
scfxm1	457	16077	14696	13145	7.8
scfxm2	914	64102	44981	43830	4.0
scfxm3	1371	145177	94355	92638	2.7
scorpion	358	5964	7750	6138	9.1
scrs8	1169	51039	38576	35810	4.8
scsd6	1350	39585	27234	25457	5.8
scsd8	2750	201559	127521	120337	2.7
sctap1	480	12212	14801	10558	9.0
sctap2	1880	144716	116299	90854	2.7
sctap3	2480	257513	202444	156835	2.0
ship04l	2118	93879	52235	57373	3.9
ship04s	1458	41269	26577	26485	5.1
ship08s	2387	98557	54658	59703	3.1
ship12s	2763	191215	104419	114655	2.5
zdeep1	164	9979	7973	7620	4.5
zdeep2	125	5908	6198	5038	5.2
zdynamico	498	78781	39802	57338	1.7
zegd051	408	13319	15296	13135	5.5
zegd085	680	37440	32780	30470	3.3
zfixmar	452	14872	13220	12209	7.7
zforestry	603	36972	24746	26964	4.3
zmeat12b	311	14091	14391	12092	6.6
zmeat31b	577	27759	21362	19601	6.1
zmeat43b	605	31446	22150	22402	5.1
zmege06	687	30884	17830	20211	3.5
zmege08	908	59110	32985	37117	2.6
zmrp3	522	9309	10834	8360	9.5
	Mean weighted value		9.23	10.42	3.42

more work than the other. Then the time for the DPRI algorithm is (instead of eqn (5.9)) :

$$K * [2*t_{lnk} + (t_{cop}+t_{snd})*(n_{dv}+n_{dj}) + t_{cop}*(n_{dj}+n_{dv})] +$$

$$(T_{SPRICE} / 2)*(1+L/100)$$

Using this modified time equation with L = 10%, the inferred times of the DPRI algorithm for our test problems are given in Table 5.6. The results suggests that dynamic load balancing might improve the algorithm's performance. However, the question of whether the improvement would actually offset the overhead cost of dynamic load balancing, requires more investigation.

5.4 Distributed CHUZR and UPDATE

5.4.1 Distributed CHUZR and UPDATE Algorithms

As stated in the introduction to this chapter, the distributed CHUZR (DCHUZR), distributed UPDATE (DUPDT) and distributed PRICE algorithms are examples of *homogeneous* distributed algorithms, since each of them involves the simultaneous execution of the *same* task on different data. While the DPRI algorithm involves the partitioning of columns among processors, the rows are partitioned in the DCHUZR and DUPDT algorithms.

Let R represent the set of rows and let $R_0, .. , R_N$ be N+1 disjoint partitions of

the set R. We use N+1 processors in a master-slave fashion. Processor 0, called the master, performs CHUZR and UPDATE on the rows in R_0 and also dispatches processors 1 thru N (the slaves) to perfrom CHUZR and UPDATE on the rows in R_1 thru R_N respectively. As discussed in §4.5, two tasks are to be accomplished by CHUZR. The first (threshold setting) is to set the threshold on the increase of the value of the entering variable. The second task, meant for numerical stability, is to select the row with the maximum pivot element among the rows yielding the same threshold value (pivot row selection). Let DCHUZR(R_p, TH_p, P_p, 0), running on processor p, represent the process of setting TH_p to the threshold obtained by considering only the rows in R_p. Let DCHUZR(R_p, TH_p, P_p, 1) be the process of setting P_p to the pivot row index obtained by considering only the rows in R_p. Let DUPDT(R_p, Pivot row), running on processor p, represent the process of updating the right hand sides of the rows in R_p. Algorithm 5.7 presents the details of the DCHUZR-DUPDT algorithms.

Algorithm 5.7 :

SX /* On the master processor */

A. For each iteration i

1. FORMC, BTRAN, PRICE, FTRAN

2. Dispatch DCHUZR(R_p, TH_p, P_p, 0) for p = 1, ... ,N

3. CHUZR(R_0, TH_0, P_p, 0)

4. Wait for DCHUZR(R_p, TH_p, P_p, 0) for p = 1, ... ,N and

then set TH_0 to the threshold obtained by considering all the

rows (set of R rows).

5. Dispatch DCHUZR(R_p, TH_0, P_p, 1) for p = 1, ... ,N

6. CHUZR(R_0, TH_0, P_p, 1)

7. Wait for DCHUZR(R_0, TH_p, P_p, 1) for p = 1, ... ,n and then set P_0 to the pivot row obtained by considering all the rows (set of R rows).

8. WRETA

9. Dispatch DUPDT(R_p, Pivot row)

10. UPDATE(R_0, Pivot row)

11. Wait for DUPDT(R_p, Pivot row)

Endfor

DCHUZR(R_p, TH_p, P_p, Par) /* On the slave processor, p ($1 \le p \le N$) */

1. Wait to be dispatched

2. Call CHUZR(R_p, TH_p, P_p, Par).

3. Signal completion and goto (1).

Subroutime CHUZR(R_p, TH_p, P_p, Par)

1. If (Par = 0) determine the threshold among the rows in R_p, set TH_p to that threshold and then RETURN.

2. If (Par = 1) determine the pivot row among the rows in R_p, set P_p to the pivot row index and then RETURN

DUPDT(R_p, P_p) /* On the slave processor, p ($1 \leq p \leq N$) */

1. Wait to be dispatched

2. UPDATE(R_p, P_p)

3. Signal completion and goto (1).

Subroutine UPDATE(R_p, Pivot row)

Update the solution vector for the rows in R_p.

5.4.2 Time Taken by DCHUZR and DUPDT Algorithms

The details concerning the implementation of the algorithms are similar to that of the DPRI implementation and is therefore not discussed separately. We now derive a formula for the time taken by the algorithms in a shared memory computing environment. The time to identify the pivot row and to update the solution vector using the above algorithms consists of the following three components : (a) the communication time; (b) the synchronization; and (c) the time, $(T_{SCHUZR}+T_{UPDATE}) / (N+1)$ where T_{SCHUZR} and T_{UPDATE} are the serial times for CHUZR and UPDATE. The synchronization tasks are steps 4, 7 and 11 of the SX program of Algorithm 5.7. Assuming that the load among the processors is perfectely balanced, the wait time in these steps is negligible. Also, in these steps, the logic for fixing TH_0 and P_0 is not computationally significant. Hence, the time for the synchronization tasks is negligible. As in the DINV and DPRI algorithms,

communication is required when the master must dispatch the slave and when the slave must signal the completion of its task to the master. These are done by calling M_FORK, M_SYNC and M_PARK. In DCHUZR, since subprocesses are spawned twice (once for setting the threshold and once for fixing the pivot row), the communication required by the algorithms can be summarized as follows. The communication for DUPDT are also given below.

	For DCHUZR	For DUPDT
# of M_SYNC's	2	1
# of M_FORK's	2	1
# of M_PARK_PROCS	2	1

Let K be the total number of iterations in the run. Let t_{sync}, t_{prk} and t_{frk} be the time to execute an M_SYNC, an M_PARK_PROCS and an M_FORK call. Then the time for fixing the pivot row and for updating the solution vector using the DCHUZR and DUPDT algorithms is given by :

$$(T_{SCHUZR} + T_{SUPDATE}) / (N+1) + 3K*[\ t_{sync} + t_{prk} + t_{frk}\ ] \quad (5.11)$$

The total serial run time of CHUZR and UPDATE for our test problems is in column 3 of Table 5.7. Using (5.11), the time taken by DCHUZR abd DUPDT to select the pivot row and update the solution vector is predicted and this prediction appears in column 4 of Table 5.7. For our problems, the inferred weighted savings with respect to the serial run time of the problem is 4.7%. The savings are given by : actual serial CHUZR and UPDATE time - inferred DCHUZR and DUPDT time.

Table 5.7 : Results with the DCHUZR and DUPDT algorithms

Problem	# of rows	Serial time for CHUZR and UPDATE	Inferred time for DCHUZR and DUPDT
agg	489	4306	3359
agg2	517	5295	4088
agg3	517	5447	4200
bandm	306	22610	21043
brandy	221	10219	10924
israel	175	4839	5624
sc205	206	3503	3912
scagr25	472	25326	19746
scfxm1	331	11754	10836
scfxm2	661	46036	32900
scfxm3	991	102108	65796
scorpion	389	9076	7832
scrs8	491	31556	24697
scsd6	148	4432	6464
scsd8	398	31751	26802
sctap1	301	9506	9181
sctap2	1091	98149	62053
sctap3	1481	179797	107422
ship04l	403	18452	15841
ship04s	403	12148	10439
ship08s	779	34191	23432
ship12s	1152	87117	54494
zdeep1	101	2178	3546
zdeep2	101	1861	2992
zdynamico	678	83286	57789
zegd051	869	40199	26796
zegd085	1443	113040	67905
zfixmar	325	10858	10073
zforestry	402	22313	18798
zmeat12b	381	13281	11645
zmeat31b	384	12908	11449
zmeat43b	648	26563	19168
zmege06	1011	26014	16724
zmege08	1343	49353	29987
zmrp3	301	7783	7627

Mean inferred weighted savings 4.69

5.5 Distributed BTRAN

In phase 1 of the revised simplex method the basic cost vector is known only after executing FORMC, whereas in phase 2, it is known immediately after the selection of the pivot row (CHUZR). Since BTRAN needs only the basic cost vector, in the distributed BTRAN algorithm BTRAN is started right after CHUZR is completed. Like the other algorithms introduced in this work, two processors are used in master-slave fashion. A flow chart of the algorithm appears in figure 5.8. A design of the implementation of the algorithm on shared memory MIMD computers is in Algorithm 5.8.

Algorithm 5.8 :

S X

Shared Storage : DATA, ETA, BI, DV, DV1

/*

DATA is for storing the data. ETA is for storing the eta file. BI contains the basis indices and DVx contains the dual vector.

*/

1. Read data into DATA
2. Call M_SET_PROCS(2)
3. For each iteration i
    a. FORMC and if i is 1 goto c
    b.
        i. If solution is infeasible goto c
        ii. Call M_SYNC
        iii. Call M_PARK_PROCS

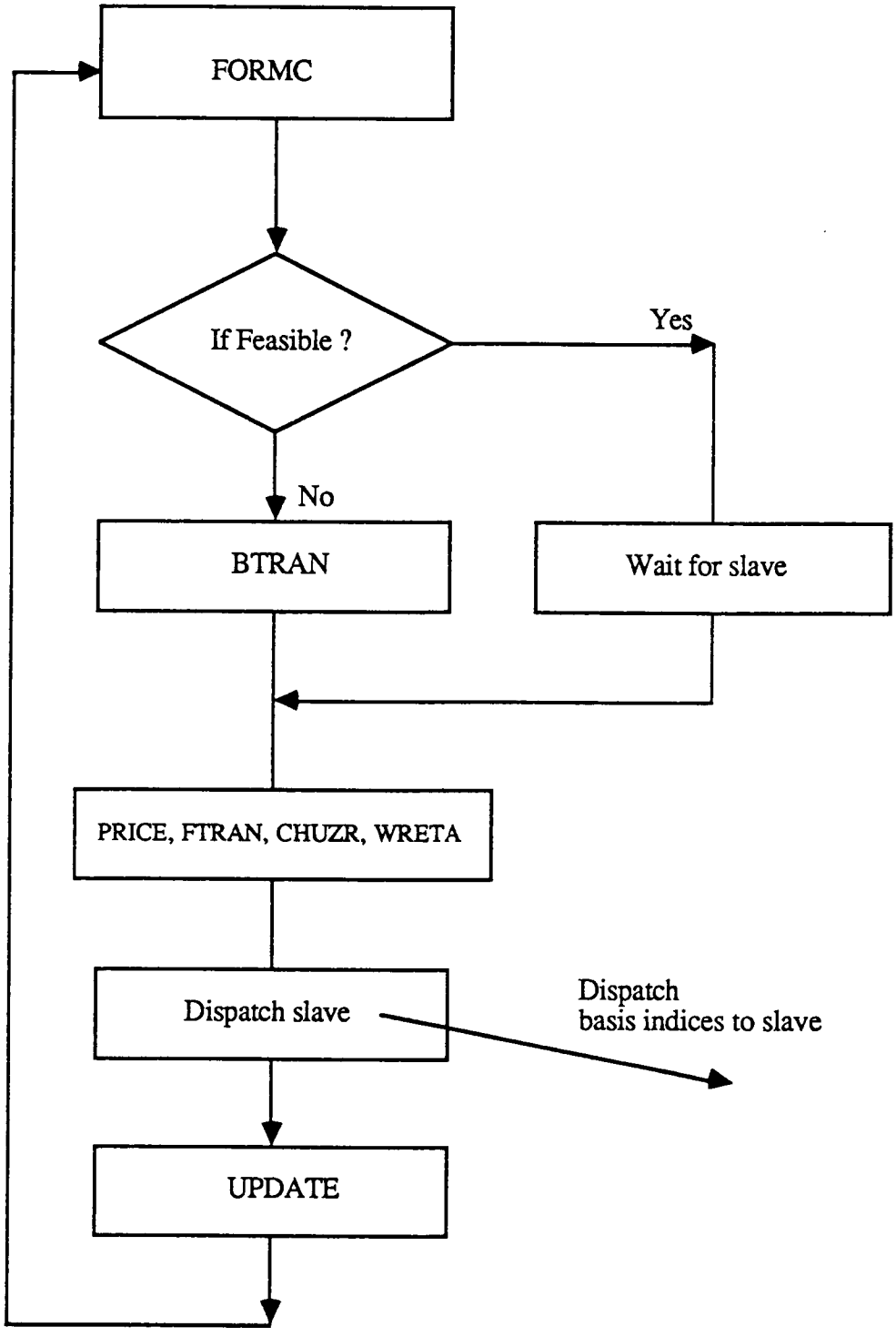


Figure 5.8 : Flow chart of the DBTRAN algorithm

iv. Copy DV1 to DV

v. Goto d

c. Call BTRAN(DATA, ETA, BI, DV) and

if i is not 1 call M_SYNC() and M_PARK_PROCS()

d. PRICE, FTRAN, CHUZR, WRETA

e. Call M_FORK(*DBTRAN*)

f. UPDATE

Endfor

B. Terminate

DBTRAN

Shared Storage : DATA, ETA, BI, DV, DV1

1. Get process ID and if 0 RETURN. Else proceed

2. Call BTRAN(DATA, ETAS, BI, DV1)

3. Call M_SYNC

4. RETURN

Subroutine BTRAN(DATAb, ETAb, BIb, DVb)

1. Compute the dual vector and put it in DV1

2. RETURN

In the DBTRAN algorithm, while the slave is executing BTRAN, the master executes UPDATE and FORMC. It then waits for the slave to complete BTRAN if the current solution is feasible. Thus, the DBTRAN algorithm saves on the UPDATE and FORMC times during the phase 2 iterations (assuming BTRAN takes more time than the total time of UPDATE and FORMC). Let f_{ph2} be the fraction of phase 2 iterations out of a total of K iterations. Let T_{FORMC}, T_{SBTRAN} and $T_{SUPDATE}$ be the times

for FORMC, BTRAN and UPDATE respectively. Then, the time for computing the dual vector using the DBTRAN algorithm is given by (5.12).

$$T_{DBTRAN} = T_{SBTRAN} - f_{ph2}*K* [T_{FORMC} + T_{SUPDATE}] + K * [t_{sync}+t_{prk}+t_{frk}] \quad (5.12)$$

CHAPTER 6

FUTURE WORK

In this work we developed revised-simplex-based algorithms to solve linear programs in a distributed computing environment. The algorithms (distributed INVERT, distributed PRICE, and so on) consisted of distributing the tasks of the revised simplex method, such as INVERT, PRICE, CHUZR, UPDATE and BTRAN, over multiple processors. We also developed mathematical characterizations of how time was spent by the revised simplex algorithm and its modules. These characterizations, referred to as timing models, provided insight into the behavior of the revised simplex method and were useful in predicting the speedups of our revised-simplex-based algorithms. Over our set of test problems, the timing models predicted that by using all of our algorithms the solution of a problem could be speeded up by an average of (nearly) 39% of the serial time. The speedup obtained by using distributed INVERT and distributed PRICE was predicted to be 34%. This led to the actual implementation of these two algorithms on the iPSC/2. Empirical results with these algorithms showed that the (approximate) speedups ranged from 5% to 41% and is 31% on the average. Thus, our work indicates that the use of distributed computers is a viable approach for revised simplex-based-solution of linear programs.

The insight gained from this work can be used in related areas of research. In this chapter, we provide a brief outline of these topics.

(a) Application to distributed Dantzig-Wolfe decomposition.

The implementation of the Dantzig-Wolfe decomposition algorithm (Dantzig and Wolfe (1960)) on distributed computers typically consists of solving the master problem and then solving the subproblems concurrently on multiple processors. For instance, Ho, Lee and Sundarraj (1988) or Lee (1988) or Ho and Gnanendran (1989). Lee (1988) (and Ho and Gnanendran (1989)) indicate that when the the time to solve the master is significantly higher than that for the subproblems, the efficacy of distributed decomposition diminishes.

The following method might improve this efficiency. When the master problem is being solved, the processors solving the subproblems are idle. Instead of letting them wait, we can use them to solve the master problem. The master processor executes the simplex iterations and the other processors can run the distributed algorithms introduced in this work. This way, the solution of the master is speeded up and the efficiency improves. Consider an example with 10 subproblems. For simplicity, assume that time for the every subproblem and the master problem is the same. If the solution of the master problem is speeded up by 20%, then the overall speedup of distributed decomposition improves to 6 (from 5.5) and the efficiency improves to nearly 55% (from 50%). The significant point is that the improvement in overall speedup and efficiency is being achieved at no extra cost since no additional processors are necessary for this method.

(b) *Implementation of distributed algorithms on shared memory systems*

In this work we implemented the distributed INVERT and distributed PRICE algorithms on the iPSC, a distributed memory MIMD computer. One of the limitations of those implementations is the communication time required for the algorithms. For

instance, communication time for the DPRI algorithm using only two processors is about 3.5% of the serial time. This time increases with the use of more processors. However, communication is fast on a shared memory computing environment. Thus, the full realization of our algorithms' potential is possible.

(c) *Prediction of an economic inversion frequency*

In chapter 4, we demonstrated that the iteration times between two consecutive reinversions could be estimated by a linear model. Based on this model, we obtained an economic value for the inversion frequency. Over our set of test problems, this inversion frequency reduced the run time by an average of 7.18%. Thus, the knowledge of the economic inversion frequency (EIF) can help solve problems more efficiently. We propose here a method to predict the EIF before solving the problem.

The Economic Inversion Frequency, k* is (cf. §4.2.2) :

$$k^* = SQRT(2I/t_i) \tag{6.1}$$

where SQRT(x) is the square root of x, I is the time for executing INVERT (reinversion procedure) and t_i is the growth in the iteration times in a INVERT cycle. Both I and t_i depend on two factors : (a) problem size, in terms of the number of rows; and (b) density and nonzero structure of the problem.

The reinversion of an r-row matrix requires an $O(r^3)$ algorithm [cf. Chvatal (1983)] and therefore its time (I) can be given by (6.2).

$$I = k_3 r^3 + k_2 r^2 + k_1 r + k_0 \tag{6.2}$$

where k_i's are constants, appropriately chosen to fit the inversion times. The exact effect of problem size on the growth rate of the iteration times is not known. Suppose we use an nth degree polynomial, then t_i is given by (6.3)

$$t_i = p_n r^n + \ldots + p_0 \tag{6.3}$$

The second factor (density and nonzero structure) will vary for different class of problems. But, problems derived from the same LP model (same class) would conceivably have similar densities and nonzero structures. For instance, the density of the problems in the SCTAP-class is almost the same. It is 4.27 for SCTAP1, and 4.3 for both SCTAP2 and SCTAP3. Therefore, we conjecture that for a given class of problems, k_i's and p_i's would be constant. That is, the values of k_i and p_i for a large problem would not differ drastically from that for a smaller problem in the same class.

The above discussion suggests that, for a given class of problems, there might be a correlation between problem size and economic nversion frequency. Based on this conjecture, the following method can be used to predict the economic inversion frequency.

1. For a given class of problems, select small problems that are representative of the class in terms of structure and density. Measure the economic inversion frequency.
2. Fit a curve to these inversion frequency and use that curve to infer the economic inversion frequency of the large problem.

BIBLIOGRAPHY

BIBLIOGRAPHY

G. Amdahl, "The validity of single processor approach to achieving large scale computing capabilities," *AFIPS Proceedings,* 30 (1967) 783-785.

G.Barnes, R.Brown, M.Katz, D.Slotnick and R.Stroker, "The Illiac IV computer," *IEEE Transactions on Computers,* C-17, (1968) 746-757.

R.H.Bartels and G.H.Golub, "The simplex method for linear programming using LU decomposition," *Communications of the ACM,* 12 (1969) 266-268.

E.M.L.Beale, "Sparseness in linear programming," in *Large sparse sets of linear equations,* J.K.Reid ed. Academic Press, London, (1971), 1-15.

J.E.Beasley, "Linear programming on Cray supercomputers," Technical Report, Department of Management Science, Imperial College, London, England, (1989).

M.Benichou, J.M.Gaithier, G.Hentges and G.Ribiere, "The efficient solution of large scale linear programming problems : some algorithmic techniques and computational results," *Mathematical Programming,* 13 (1977) 280-332.

R.E.Bixby, "Vectorizing a protable C implementation of the simplex method," Presented at the ORSA/TIMS joint national meeting, Vancouver, Canada, (1989).

V.Chvatal, *Linear Programming,* W.H.Freeman (1983).

A.M.Costa and L.P.Jennergren, "Trade and development in the world economy : methodological features of project DYNAMICO," *Journal of Policy Modeling,* 4 (1982) 3-22.

G.B.Dantzig, *Linear Programming and Extensions,* Princeton University Press, Princeton, (1963)

G.B.Dantzig and W.Orchard-Hays, "The product form for the inverse in the simplex method," *Mathematical Tables and Other Aids to Computation,* 8 (1954) 64-67.

G.B.Dantzig and P.Wolfe, "Decomposition principle for linear programs," *Operations Research,* 8 (1960) 101-111.

D.Dewitt, R.Finkel and M.Solomon, "The CRYSTAL multicomputer : design and implementation experience," *IEEE Transactions on Software Engineering,* 8 (1987) 953-966.

J.J.Dongarra and E.Grosse, "Distribution of mathematical software via electronic

mail," *Communications of the ACM*, 30, (1987) 403-407.

J.J.Dongarra and S.C.Eisenstat, "Squeezing the most out of an algorithm in Cray Fortran," *ACM Transactions on Mathematical Software*, 10 (1984) 221-230.

EPRI, "Decomposition of linear programs using concurrent processing on multicomputers," Electric Power Research Institute Report RP1999-11, April, (1989).

D.Finon, "Un modele energetique pour la France," Editions du CNRS, Paris, (1976).

M.J. Flynn, "Very high speed computing," *IEEE Proceedings*, 54 (1966) 1901-1909.

M.J.Flynn, "Some computer organizations and their Effectiveness," *IEEE Transactions on Computers*, C-21 (1972) 948-960.

J.J.H Forrest and J.A.Tomlin, "Updating triangular factors of the basis to maintain sparsity in the product form of simplex method," *Mathematical Programming*, 2 (1972) 263-278.

J.J.H Forrest and J.A.Tomlin, "Vector processing in simplex and interior methods for linear programming, "Presented at the ORSA/TIMS joint national meeting, New York, (1989)

G.C.Fox and P.C.Messina, "Advanced computing architectures," *Scientific American*, 257 (1987) 66-77

D.M.Gay, "Electronic mail distribution of linear programming test problems," *Mathematical Programming Society COAL Newsletter*, December (1985).

D.Gelernter, "Programming for advanced computing," *Scientific American*, 257 (1987) 90-115

R.V.Helgason, J.L.Kennington and H.A.Zaki, "A parallelization of the simplex methods," Technical Report, Southern Methodist University, Dallas, TX (1987).

M.T.Heath, T.H.Dunigan and C.H.Romine, "Parallel computing survey," Unpublished tutorial sessions, Oak Ridge National Laboratories, TN (1987).

E.Hellerman and D.Rarick, "Reinversion with the preassigned pivot procedure," *Mathematical Programming*, 1 (1971) 195-216.

E.Hellerman and D.Rarick, "The partitioned preassigned pivot procedure," in *Sparse Matrix and Their Applications*, D.J.Rose and R.A.Willoughby, eds, New York, Plenum Press, (1972) 67-76.

W.D.Hillis, *The Connection Machine*, MIT Press, Cambridge, MA, (1985).

J.K.Ho, "Optimal design of multi-stage structures : a nested decomposition

approach," *Computers and Structures*, 5 (1975) 249-255.

J.K.Ho and S.K.Gnanendran, "Distrtibuted decomposition of block-angular linear programs on a hypercube computer," Technical Report, Management Science Program, University of Tennessee, Knoxville, TN, (1989).

J.K.Ho and E.Loute, "A set of staircase linear programming test problems," *Mathematical Programming*, 20 (1981) 245-250.

J.K.Ho, T.C.Lee and R.P.Sundarraj, "Decomposition of linear programs using parallel computation," *Mathematical Programming*, 42 (1988) 391-405.

J.K.Ho and R.P.Sundarraj, *DECOMP : An Implementation of Dantzig-Wolfe Decomposition for Linear Programming*, Springer-Verlag, New York, (1989)

R.Hockney and C.Jeeshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, England, (1981).

K.Hwang and F.A.Briggs, *Computer Architecture and Parallel Processing*, McGraw Hill, New York, (1984).

IBM Corporation, "MPSX/370 Version 2 Program Reference Manual," Order No. SH19-6553, (1988).

Intel Corporation, "iPSC/2 Fortran Programmer's Manual," Order No. 311019-003, (1988a)

Intel Corporation, "iPSC/2 Green Hills Fortran Language Reference Manual (Preliminary)," Order No. 311020-003, (1988b).

A.Karp, "Programming for Parallelism," *Computer*, 20 (1987) 43-57.

J.S.Kowalik, *High Speed Computation*, NATO ASI Series, Computer and System Sciences 7, Springer-Verlag, Berlin, (1984).

D.J.Kuck, "Illiac IV software and application programming," *IEEE Transactions on Computers*, C-17 (1968) 758-770.

T.C.Lee, "Distributed optimization of linear programs using Dantzig-Wolfe decomposition," PhD dissertation, University of Tennessee, Knoxville, (1988).

I.J.Lustig, "An analysis of an available set of linear programming test problems," *Computers and Operations Research*, 16 (1989) 173-184.

A.S.Manne, "Sufficient conditions for optimality in an infinite horizon development plan," *Econometrica*, 38 (1970) 18-38.

A.S.Manne, "U.S. options for the transition from oil and gas to synthetic fuels," Discussion Paper No. 26D, Public Policy Program, John F. Kennedy School of Government, Harward University, January, (1975).

H.M.Markowitz, "The elimination form of inverse and its application to linear programming," *Management Science*, 3 (1957) 255-269.

W.A.McKenney, "Dantzig-Wolfe decomposition based technique for capacitated material requirement planning," Ph.D dissertation, Management Science Program, University of Tennessee, Knoxville, (1988).

D.Medhi, "Decomposition of structured large scale optimization problems and parallel optimization," Technical Report No. 718, Computer Sciences Department, University of Wisconsin, Madison, WI, (1987).

D.K.Merchant and G.L.Nemhauser, "A model and an algorithm for the dynamic traffic assignment problems," *Transportation Science*, 12 (1978) 183-199.

B.A.Murtagh, *Advanced Linear Programming : Computation and Practice*, McGraw Hill, New York, (1981).

B.A.Murtagh and M.A.Saunders, "MINOS/AUGMENTED User's Manual," Report SOL 80-14, Stanford University, Stanford, CA, (1980).

K.W.Neves, "Vectorizing of scientific software," in Kowalik (1984) 271-291.

B.Noble and J.W.Daniel, *Applied Linear Algebra*, Prentice-Hall Inc., Englewood Cliffs, NJ, (1977).

W.Orchard-Hays, *Advanced Linear Programming Computing Techniques*, McGraw Hill, New York, (1968).

J.M.Ortega and R.G.Voigt, "Solution of partial differential equations on vector and parallel computers," *SIAM Review*, 27 (1985) 149-240.

U.W.Pooch and A.Nieder, "A survey of indexing techniques for sparse matrices," *Computing Surveys*, 5 (1973) 109-133.

U.Schendel, *Introduction to Numerical Methods for Parallel Computers*, Ellis-Horwood, Chrichester, England, (1984).

J.Schwartz, "A taxonomic table of parallel computers based on 55 designs," Ultracomputer Note No. 69, Courant Institute, New York University, NY, (1983).

C.L.Seitz, "The cosmic cube," *Communications of the ACM*, 28 (1985) 22-33.

Sequent Corporation, "Balance Guide to Parallel Programming," Order No. 1003-43311, (1986).

H.S.Stone, *Introduction to Computer Architecture*, Science Research Associates, Inc., Chicago, IL, (1982).

J.C.Stone, E.S.Klotz and C.J.Strauss, "WHIZARD for vector processors : a progress report," Working Paper, Ketron Management Science Inc., 1700 North Moore

Street, Arlington, VA, (1989).

W.Swart, C.Smith and T.Holderby, "Expansion planning for a large diary farm," in H.M.Salkin and J.Saha (eds.), *Studies in Linear Programming*, North-Holland, Amsterdam, (1975) 163-182.

J.A.Tomlin, "LPM1 user's guide," Unpublished manuscript, Systems Optimization Laboratory, Stanford University, Stanford, CA, (1973).

S.A.Zenios, "Parallel numerical optimization : current status and an annotated bibliography," *ORSA Journal on Computing*, 1 (1989) 20-43.

S.A.Zenios and J.M.Mulvey, "Nonlinear network programming on vector supercomputers : a study on the Cray X-MP," *Operations Research*, 34 (1986) 667-682.

S.A.Zenios and J.M.Mulvey, "Vectorization and multitasking of nonlinear network programming algorithms," *Mathematical Programming*, 42 (1988) 449-470.

APPENDIX

TEST PROBLEMS

In this chapter we describe the test problems used throughout the work. We have used a total of 35 medium-sized problems as our test set. The problems sizes range from 205 to 1481 rows, 163 to 2763 columns and 552 to 11334 nonzeros. All the problems are of the form :

$$\begin{aligned} \min \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \\ & \mathbf{Ax} \lessgtr \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

where the symbol <> indicates that the problem can have equality, less-than-or-equal or greater-than-or-equal constraints. Table A.1 shows a listing of the test problems, along with their sizes.

Gay (1985) has collected LP test problems from many sources and made them publicly available via Netlib [Dongarra (1987)]. A substantial number of the test problems used in this work can be obtained from Netlib and they have been identified in the table. Computational results using MINOS [Murtagh and Saunders (1980)] with problems from Netlib have been reported by Lustig (1989).

All the SCxxx problems have staircase structure [Ho and Loute (1981)]. SC205 is a dynamic multisector development planning model adapted from Manne (1970). SCAGR25 is a version of a large dairy farm expansion planning model described in Swart, Smith and Holderby (1975). SCRS8 is a technological assessment model for

Table A.1 : Listing of test problems

Problem	# of rows	# of structural columns	# of structural nonzeros	Application area
agg	489	163	2541	†
agg2	517	302	4515	
agg3	517	302	4531	
bandm	306	472	2659	
brandy	221	249	2150	
israel	175	142	2358	
sc205	206	203	552	MultiSctDev
scagr25	472	500	2029	Dairy
scfxm1	331	457	2612	Prodn Sch
scfxm2	661	914	5229	Prodn Sch
scfxm3	991	1371	7846	Prodn Sch
scorpion	389	358	1708	Dyn Energy
scrs8	491	1169	4029	Tech Assmt
scsd6	148	1350	5666	Struc Dsgn
scsd8	398	2750	11334	Struc Dsgn
sctap1	301	480	2052	TAP
sctap2	1091	1880	8124	TAP
sctap3	1481	2480	10734	TAP
ship04l	403	2118	8450	
ship04s	403	1458	5810	
ship08s	779	2387	9501	
ship12s	1152	2763	10941	
zdeep1	101	164	4982	Random
zdeep2	101	125	4220	Random
zdynamico	678	498	5184	World Trade
zegd051	869	408	2181	Elec Dsptch
zegd085	1443	680	3635	Elec Dsptch
zfixmar	325	452	2601	
zforestry	402	603	3794	
zmeat12b	381	311	2992	MeatIngrMix
zmeat31b	384	577	4284	MeatIngrMix
zmeat43b	648	605	4726	MeatIngrMix
zmege06	1011	687	6066	ElecCapExp
zmege08	1343	908	8200	ElecCapExp
zmrp3	301	522	2010	MRP

† Blank indicates that application area is not known

the transition from fossil to renewable energy resources in the U.S. [Manne (1970)]. SCORPION [Finon (1976)] is a dynamic energy flow model developed for the oil sector of France. SCSDx are sample problems in the minimal weight design of multistage trusses under a single loading condition. This model has been developed by Ho (1975). SCFXM1 is a real world problem in production scheduling. SCFXM2 and SCFXM3 are hypothetical extensions of SCFXM1. The origin of these three problems is not known. SCTAPx are problems in the optimization of the dynamic flow of traffic network where congestion is modeled explicitly in the flow equations. This model was developed by Merchant and Nemhauser (1978).

The remaining problems have block-angular sructure. ZDEEPx are randomly generated problems with dense blocks. ZMEATxx are sample problems for multiproduct ingredient mix optimization in the meat industry. ZFORESTRY is from a forest management model. ZDYNAMICO is based on a model for world trade and development from the United Nations, by Costa and Jennergren (1982). ZMRP3 is a materials requirement planning model given in McKinney (1987). ZEGDxxx are problems in the economic dispatch of electric power generators to satisfy system constraints imposed by, for example, regulation and reserve margin requirements. ZMEGExx is a problem in multiregional electric capacity expansion planning. Both these models (ZEGD and ZMEGE) are described in EPRI (1989).

VITA

The author R. P. Sundarraj was born in Madras, India on September 15 1960.

He earned a Bachelor's degree in Electronics and Communications Engineering from the University of Madras, India in 1982 and a Master's degree in Management Science from the University of Tennessee, Knoxville in 1988. He expects to receive a doctoral degree in Management Science in August 1990. He is currently employed as Assistant Professor of Management in Clark University, Massachusetts.