Parallel Execution of the Simplex Algorithm

Sungwoon Choi

Parallel Execution of the Simplex Algorithm

by

Sungwoon Choi

A research project submitted in partial fulfillment of
the degree of Master of Science

Major Professor
Dr. T. G. LEWIS

Department of Computer Science
Oregon State University
Corvallis, Oregon

April 30, 1988

Acknowledgements

I am grateful to my advisor, Professor Ted Lewis, for his guidance, understanding, helpfulness and encouragement. His experience and assistance have been instrumental at innumerable points in the progression of this problem. I would also like to thank my parents for their support, as well as my wife, Yejung, and my daughter, Eunyung, for their patience, help and understanding.

Sungwoon Choi
May, 1988

Table of Contents

List of Figures

PARALLEL EXECUTION OF THE SIMPLEX ALGORITHM

ABSTRACT

This project is concerned with the optimal distribution of the computation and the data in parallelized Simplex algorithms. Test cases were implemented on a 16-processor Transputer system from INMOS Corporation. By careful consideration of distribution of computations and data, a nearly linear speedup pattern was obtained. The most interesting thing in this study was that 1) the execution time is not dependant on communication delay, 2) overhead due to parallelization does not significantly increase as the number of processors increase and 3) the Simplex algorithm communication delay is not so significant if the problem size is big enough.

Keywords and phrases : message-passing, two-phase Simplex, linear programming, distributed memory, multiprocessor system

1.0 Introduction

Distributed computation has for many years been the focus of considerable research, offering a number of advantages, such as ready availability, high degree of reliability, high performance, and the ability to incrementally add to the system due to their modular structure [1]. The objective of this study is to determine the best utilization of parallel

The objective of this study is to determine the best utilization of parallel processors for solving the two-phase Simplex algorithm, which is the most popular algorithm for solving linear programming problems [2]. This problem can be addressed two ways: 1) algorithm issues, and 2) architectural issues.

Algorithm issues; Two methods may be used to implement the parallel Simplex algorithm [3], either 1) segmentation by rows or 2) segmentation by columns. The major difference between these two methods is the way they perform communication. They may yield almost identical results when run on a shared memory machine which is assumed to have no communication delay. But experiments show segmentation by rows is advantageous for message-passing communication as in the Transputer system since row segmentation reduces overall communication cost.

Architectural issues; Compute-Aggregate-Broadcast algorithms are composed of three phases: 1) a compute phase which performs some basic computation, 2) an aggregate phase which combines local data into one or a few global values, and 3) a broadcast phase which returns global information back to each process [4]. A tree structure with three children may be the best means to reduce the communication overhead in Transputer systems since it has the shortest diameter, $2log_3p$, among the structures which can be built using Transputer network. But in this experiment, a sequential communication path is used, because we found no big difference in communication delay between the sequential and the tree structured architecture when the communication path is short.

In addition it is very hard to map the computation and data to a tree structured Transputer system because of the special architecture of the Transputer board. If the number of processors gets larger, a tree structured communication path may be preferred.

In this study, almost linear speedup was obtained for Simplex algorithms based on a sequential communication path, and careful distribution of computation and data.

2.0 Background

2.1 Performance Measures [3]

Notations are defined as follows :

- Execution time, **T (p,m,n)** is the time to compute a Simplex problem with m constraints and n variables on p processors.

  Execution time includes initialization and communication time but does not include disk input/output time.
- Speedup, **S (p,m,n)** is the ratio of the time required by the serial algorithm divided by T(p,m,n).

  $S(p,m,n) = T(1,m,n) / T(p,m,n)$
- Efficiency, **E (p, m, n)** = $S(p,m,n) / p$.
- Communication delay, **D (p, m, n)** is the delay time due to data communication.

2.2 The Simplex Method

The Simplex method, in conjunction with certain auxiliary procedures, provides an algorithm for the solution of standard linear programming problems.

2.2.1. Definition [2]

Linear programming can be defined as the optimization of problems such as

solve for $x_1, x_2, \ldots\ldots, x_{n-1}, x_n,$ such that

max or min $f(x_1, \ldots\ldots, x_n)$,

subject to $g(x_1, ... ,x_n) \leq \text{or} = \text{or} \geq b_i \quad i = 1, ... ,n,$

in which the objective and constraint functions are all linear.

Any given linear programming problem, after suitable algebraic manipulation, can be represented in standard form, P:

Max $z = \mathbf{c}^T\mathbf{x}$

subject to $\mathbf{A}\mathbf{x} = \mathbf{b},$

where $\mathbf{x} \geq \mathbf{0}, \mathbf{b} \geq \mathbf{0},$ and

A is an m x n matrix, representing the constraints,

x is a n element solution vector,

b is the right hand side of m constraints, and

c is the n coefficients of the objective function.

A basic feasible solution (BFS) is a non-zero solution which satisfies the constraint set $\mathbf{A}\mathbf{x} = \mathbf{b}, \ \mathbf{x} \geq \mathbf{0}$, and can be collected into an m x 1 vector $\mathbf{x}_B$. The remaining (n - m) nonbasic variables, whose values are required to be zero, are contained in the vector $\mathbf{x}_R$. The columns of **A** associated with the variables of a basic feasible solution $\mathbf{x}_B$ are assembled into the m x m basis matrix **B**. By reordering the original problem variables, along with their associated columns $\mathbf{a}_j$ and cost coefficients c_j, the original vector of variables can be partitioned into basic and nonbasic pieces: $\mathbf{x} = [\mathbf{x}_B \ \mathbf{x}_R]$. In accordance with this partition, the constraint matrix **A** then can be partitioned into **[B R]** where **B** is the basis matrix associated with $\mathbf{x}_B$ and **R** is the m x (n - m) matrix of nonbasic columns. The values of the variables for this particular BFS are given by $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}$ and $\mathbf{x}_R = \mathbf{0}$ and the associated partition of the cost vector is $\mathbf{c}^T = [\mathbf{c}_B{}^T$

$\mathbf{c}_R^T$]. Now the original problem P can be rewritten as

$$\begin{array}{ll} \text{Max} & z = \mathbf{c}_B^T \mathbf{x}_B + \mathbf{c}_R^T \mathbf{x}_R \\ \text{subject to} & \mathbf{B}\mathbf{x}_B + \mathbf{R}\mathbf{x}_R = \mathbf{b} \\ \text{and} & \mathbf{x}_B, \mathbf{x}_R \geq \mathbf{0}. \end{array}$$

And for any basis **B** and for any column $\mathbf{a}_j$ of **A**, whether or not it is a basic column, $\mathbf{y}_j = \mathbf{B}^{-1}\mathbf{a}_j$ and $z_j = \mathbf{c}_B^T\mathbf{y}_j = \mathbf{c}_B^T \mathbf{B}^{-1}\mathbf{a}_j$ can be defined. Although $\mathbf{y}_j$ and z_j by themselves have no particular name, the values of $\mathbf{y}_j$ are the scalar coefficients in the expression of $\mathbf{a}_j$ as a linear combination of the basic columns of **B** and the value $(z_j - c_j)$ can be expressed as the reduced cost of the jth column or of the jth variable.

The Simplex method is the most popular algorithm for solving the linear programming problem. Even if its time complexity is theoretically exponential in its worst case and polynomial in the expected number of iterations, practice has shown that it is capable of rapid convergence, requiring m to 3m iterations to complete[10].

2.2.2 Computational Procedures [2][10][11]

The current values of the relevant variables and quantities are stored in a skeleton diagram, called a simplex tableau, as shown in Figure 2-1. All variables, x_j, basic or not, are listed across the top and the current basic columns, $\mathbf{b}_j$, are listed by name at the left. The second column of the tableau contains the values of the basic variables and the objective function which is being maximized. Each of the remaining columns belongs to (or is associated with) an $\mathbf{x}_j$ value and contains an $\mathbf{y}_j$ vector and its corresponding $(z_j - c_j)$ value.

Figure 2-1 Simplex Tableau

		x_1	x_2	....	x_n
b_1	x_{B1}	y_{11}	y_{12}	....	y_{1n}
b_2	x_{B2}	y_{21}	y_{22}	...	y_{2n}
.	.	.	.		.
.	.	.	.		.
.	.	.	.		.
b_m	x_{Bm}	y_{m1}	y_{m2}	....	y_{mn}
	Max z	$z_1 - c_1$	$z_2 - c_2$	....	$z_n - c_n$

Assume then, that the current basic feasible solution is not optimal and that the current values of all $\mathbf{x}_{Bi}$, all $\mathbf{y}_{ij}$, and all $(z_j - c_j)$ are known. This solution can be improved by changing the value of one of the nonbasic variables x_j, i.e., pivoting to an adjacent extreme point. If the goal is to increase the value of z as fast as possible, an obvious strategy is to increase the variable, x_j, having the most negative reduced cost;

Simplex basis entry criterion : The nonbasic variable, x_k, is chosen to increase and enter the basis if and only if $(z_k - c_k)$ is negative and

$(z_k - c_k) = \min (z_j - c_j)$,

where j is an index over all nonbasic variables.

Since the variable, x_k, is chosen to enter the basis, one of the current basic variables must be ejected. But because of the nonnegativity constraint of BFS, a pivot must be selected according to the following rule.

Simplex basis exit criterion : Given that the variable x_k is to enter the basis, the column $\mathbf{b}_r$ and variable $\mathbf{x}_{Br}$ must leave,

where $\mathbf{x}_{Br}/y_{rk} = \min (x_{Bi}/y_{ik}) \;\; y_{ik} > 0$,

and i is an index of basic variables.

The next pivot consists of the variable x_k entering and x_{Br} leaving the basis. At each pivot the old values are read from the current tableau and the transformation can be performed. The newly calculated values are then entered in a new simplex tableau.

Although a fair amount of algebraic manipulation is necessary, this may be minimized and the chance of error can be reduced by arranging the computation in an orderly and symmetrical manner. For example form a new vector Φ from the tableau column belonging to x_k:

$$\Phi = (-y_{1k}/y_{rk}, \ldots, -y_{r-1,k}/y_{rk}, 1/y_{rk}-1, \\ -y_{r+1,k}/y_{rk}, \ldots, -y_{mk}/y_{rk}, -(z_k - c_k)/y_{rk})$$

Note that the last element, $(z_k - c_k)$, is treated the same way as all the y_{ik} values, with the exception of y_{rk}. In the previous tableau the element y_{rk} was placed in the column of the newly entering variable, x_k, and in the row of the departing basic variable, x_{Br}.

In the new tableau the various labels, x_j and $\mathbf{b}_i$, will be the same as in the previous tableau, except that the rth basic column $\mathbf{b}_r$ now will be $\mathbf{a}_k$. It may then be asserted that each column of entries for the new tableau is derived from the ith corresponding column in the old tableau by means of the following transformation :

(new column j) = (old column j) + $y_{rj}\,\Phi$.

Computational Procedures:

1) Select the **pivot** column k so that $c_k < 0$. If the column does not exist, stop; the optimal solution has been found.
2) select the **pivot** row r with the smallest positive ratio, b_r/y_{rk}, for $y_{rk} > 0$. If the row does not exist, stop; the objective function is unbounded.
3) Perform the transformation on matrix **A:**

```
for col := 1..n+1 do
        for row := 1..m+1 do
                $y_{row,col} := y_{row,col} + \Phi_{row} \times y_{r,col}$
```

4) Return to step (1) and repeat the procedure.

At each iteration, the Simplex method transforms the problem, increasing the objective function while observing the constraints.

2.2.3 Two Phase Simplex Method [2]

By means of the Simplex method developed in the previous section, any linear programming problem (LPP) in standard form can be solved, provided that redundant constraints have been eliminated and that a basic feasible solution has been identified. Unfortunatly, an LPP arising in the real world may not meet these requirements. Therefore to solve any standard-form LPP, an initial basic feasible solution must be grafted onto the Simplex method by a preliminary procedure that identifies redundant constraints. The preliminary procedure is phase 1 of the so-called "two-phase" Simplex algorithm.

The two-phase Simplex algorithm for solving the linear programming problem P can be summarized as follows:

1) Given the standard form of linear programming problem, the auxiliary problem Q can be formed as:

Max $z' = -\mathbf{1w}$

subject to $\mathbf{Ax} + \mathbf{I}_m\mathbf{w} = \mathbf{b}$

where $\mathbf{x} \geq \mathbf{0}, \mathbf{w} \geq \mathbf{0}$

$\mathbf{1}$ is an m-component row vector with all 1's

$\mathbf{I}_m$ is an mxm identity matrix

$\mathbf{w}$ is an m-component column vector of artificial variables

If the matrix $\mathbf{A}$ contains a submatrix $\mathbf{I}_m$, it will serve as an initial basis. Then, if there are no redundant constraints and an initial basic feasible solution has been found; proceed to step (4). If **A** does not contain an identity submatrix, produce one by adding a sufficient number of nonnegative artificial variables, w_i.

2) phase 1. Use the Simplex method to solve the problem Q. If the optimal value of the phase-1 objective function is negative then P has no feasible solution. But if the minimum value of the sum is zero, prolong phase 1 in an effort to drive all artificial variables out of the basis.

3) If all artificial variables can be expelled, then a BFS to problem P has been found. On the other hand, for every artificial variable w_s that cannot be removed from the basis, the s^{th} constraint of problem P is redundant. In either case, so long as phase 1 ends with a zero valued objective, the final tableau is converted into the initial tableau of the phase 2 simply by deleting the nonbasic artificial columns and recalculating objective value, z and all $(z_j - c_j)$ in accordance with the original objective function.

4) Phase 2. Apply the Simplex method to the modified tableau until the optimal solution to problem P is obtained. Note that an artificial variable that could not be driven out of the basis in phase 1 will remain in the basis with a zero value throughout phase 2.

2.3 Transputer Systems [5]

2.3.1 Transputer Hardware Configuration

A Transputer is a microcomputer with its own local memory and links for connecting one Transputer to another. Our test system consists of the IMS B004 IBM Personal Computer add-in board and four IMS B003 evaluation boards. The IMS B004 board has one T414 Transputer with two megabytes of RAM, and PC subsystem logic, allowing a program running on the PC to reset and analyze systems. The IMS B004 board also has an IMS C002 link adaptor, interfacing with a parallel address/data bus. Each IMS B003 board is a double extended Eurocard containing four T414 Transputers, each of which has 256 Kbytes of dynamic RAM. Thus the complete system has 16 transputers with a total of 4 Mbytes of Ram, connected to the B004 Transputer which is inside the PC.

The 1.5 micrin CMOS IMS T414 (Fig. 2-2) integrates a 32-bit microprocessor, four standard Transputer communications links, 2 Kbytes of on-chip RAM, and memory and peripheral interfacing on a single chip. Each Transputer on the IMS B003 board (Fig. 2-3) is connected in a square with rotational symmetry. Link 2 of each Transputer is connected to link 3 of the next Transputer and links 0 and 1 of each Transputer are directed to the edge connector. Because of the edge connector, it is possible to freely choose the shape of the system used.

2.3.2 Transputer Development System

Our system is designed and developed under the TDS D700C system using Occam II as its programming language. The Transputer Development System (TDS) provides the following major facilities:

* Edit, compile, and run an Occam 2 program within the system.

Figure 2-2 Transputer (T414) Block Diagram

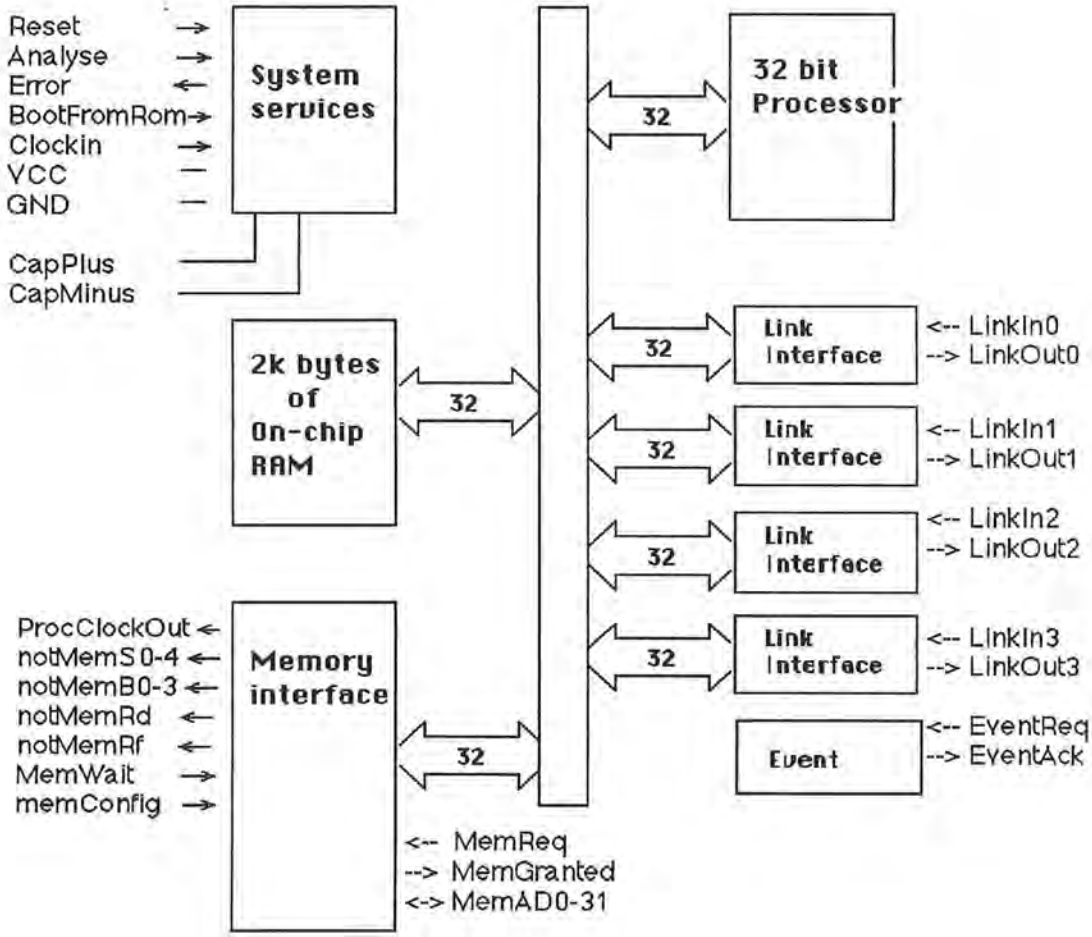
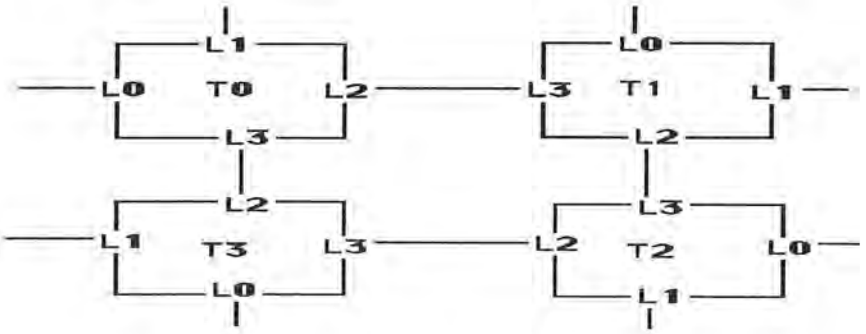


Figure 2-3 IMS B003 Evaluation Board Link Connection



- Configure an Occam program for a network of Transputers, and load it into the network from a link on the TDS's Transputer board.
- Analyse a network of running Transputers and obtain the program source line corresponding to the current process running on each processor.

2.3.3 Occam [6][8]

Occam is a high level language which provides a framework for the design of concurrent systems, using Transputers in the same way that boolean algebra provides a framework for the design of electronicsystems from logic gates. A program running in a Transputer is formally equivalent to an Occam process, so that a network of Transputers can be described directly as an Occam program.

Occam provides some special features for interprocess communication and parallelization: primitives like ! to output to a channel and ? to input from a channel and constructs like Par, which ensures that the process is executed in parallel, and Alt which observes a number of channels for the first input, then executes the process associated with that input. Communication in Occam is synchronous, unbuffered, and supported by hardware in the Transputer. To establish interprocess communication, processes must execute in parallel and share a common channel.

3.0 Experiments in parallelized Simplex method

Parallel processing can be addressed on three levels [7]: 1) Architectural issues, including the means of organizing and implementing computers with multiple processing elements; 2) Software issues, including applications and system software; and 3) Algorithm issues, which are concerned with the problem of parallel algorithm design. In this section, only the architectural and algorithm issues for implementation of the two-phase simplex method in a network of Transputers are considered.

3.1 Architectural Issues

A Transputer network can be easily transformed into different structures, eg., mesh, shuffle or other structures with the limitation that each processor has only four links, two of which are already linked internally in one board. Each board is limited to four processors.

The Transputer network was configured as a tree with three children (Fig. 3-1) to reduce communication overhead on the first implementation. The Simplex method requires a controller processor, which is responsible for overall system synchronization (see section 2.2). For this mechanism, an architecture with the shortest diameter is most suitable in the sense of execution time, but less desirable from the development and maintenance point of view. The tree structure with three children is hard to map into a real system and the programs can not be generalized. In this instance, the tree structured network was

Figure 3-1 Tree structured Transputer Network with Three Children

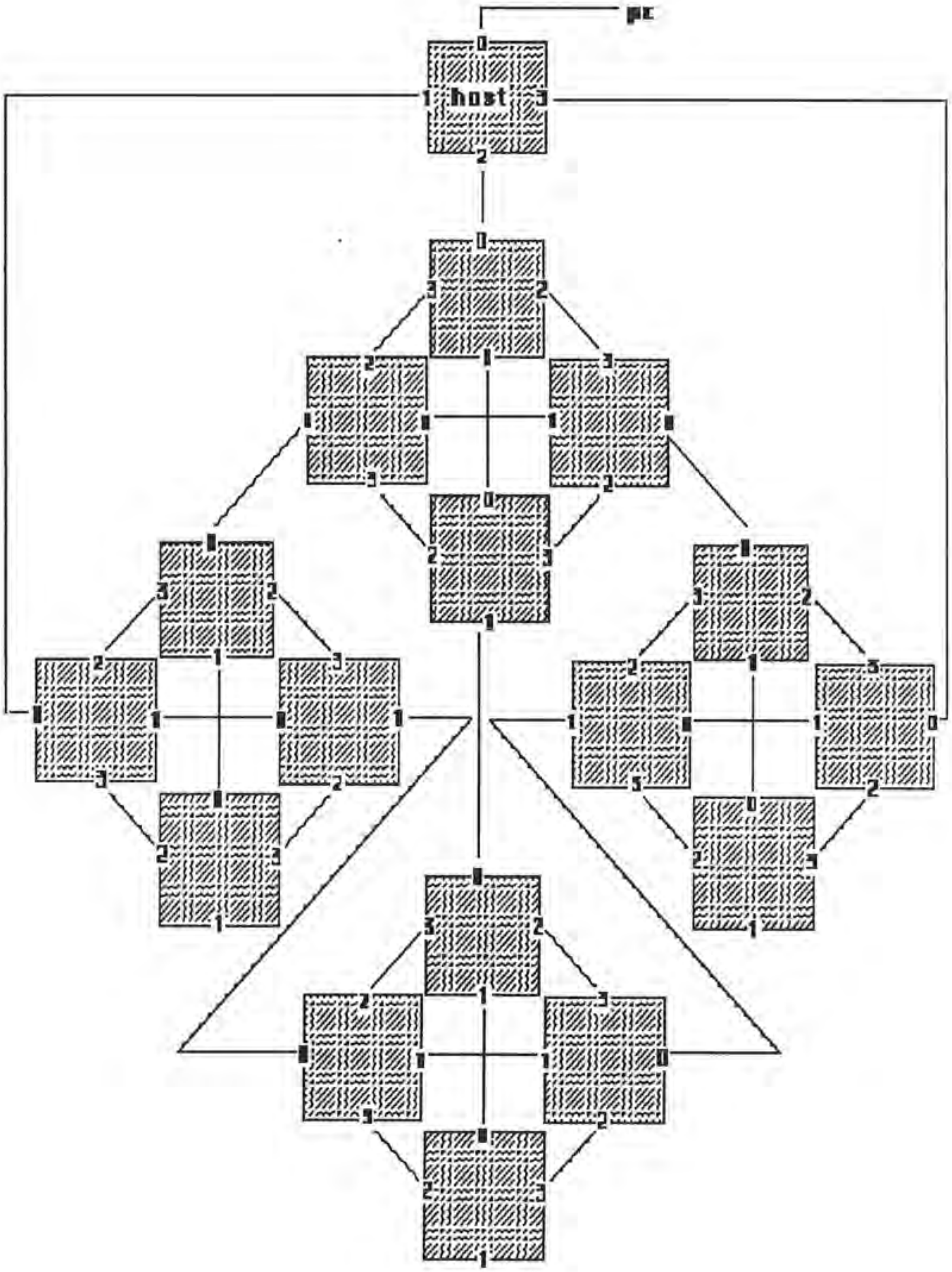
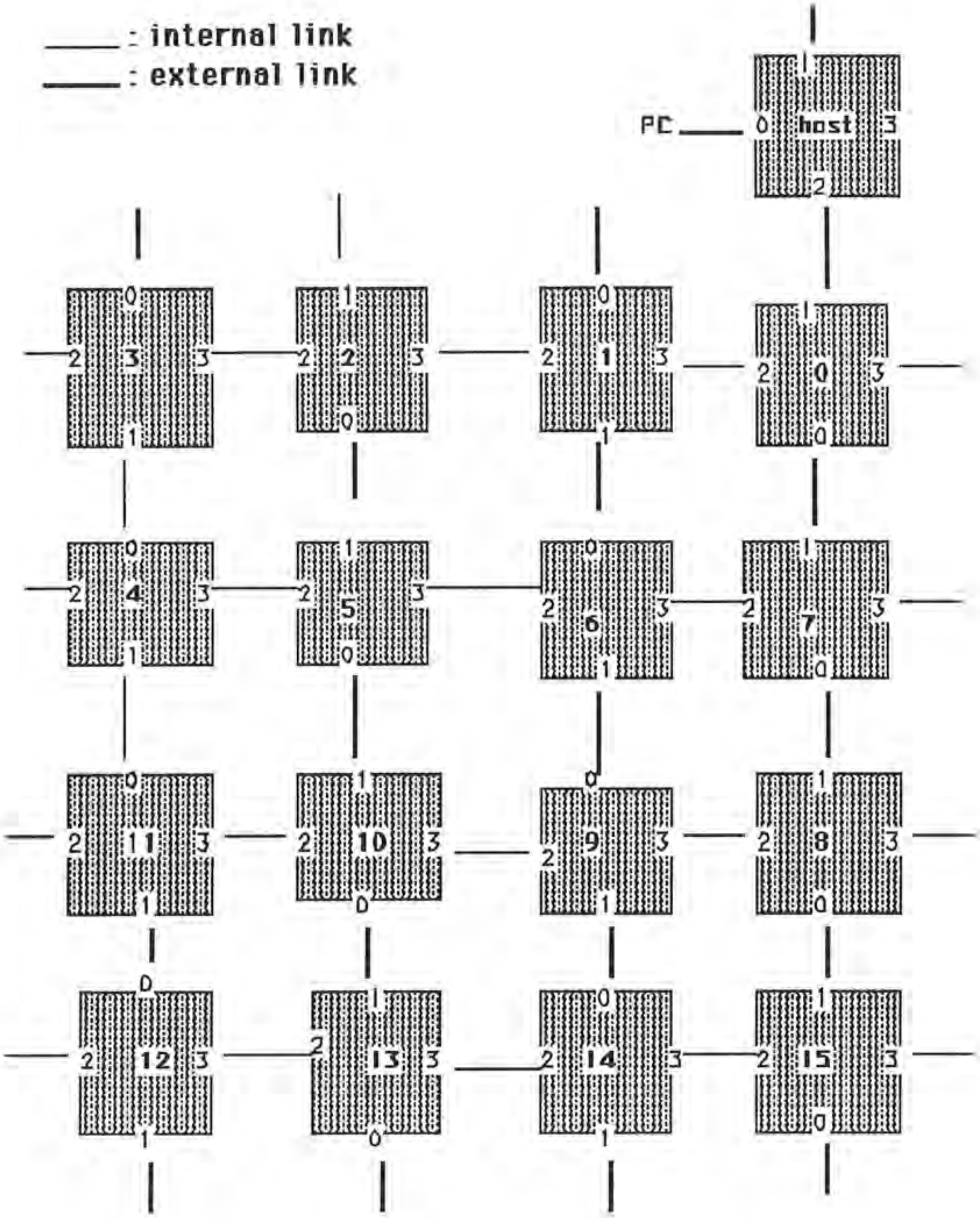


Figure 3-2 Mesh structured Transputer Network



simulated by a sequential Transputer network (Fig. 3-2). In a large Transputer network, a tree structure will be required, but for a system with 16 processors such as the one in question, there is not a great difference in communication overhead between the tree and the sequential network. This is shown by the test cases run in section 3.3.

3.2 Algorithm Issues

The Simplex method can be parallelized in two fundamentally different ways. One is to parallelize the pivoting process by Simplex method step (1) and reduce the number of iterations. Another method is to distribute the computation and data in accordance with Simplex method step (3), which is the means of implementation used in this study. There are two alternatives, distribution by rows (Fig. 3-3) and distribution by columns (Fig. 3-4), which yield nearly identical results. However the former method involves less communication overhead in step(1) and step(2) of the Simplex method and can reduce the effort of redistributing the matrix in phase 2 of the Simplex method (see section 2.2).

3.2.1. Distribution by rows

Fig. 3-5 shows the Task Graph [9] of the Simplex method distributed by rows in the Transputer network. Tasks in the same columns represent processes executed by the same processor, which is made clear by the zero communication delays between tasks. Tasks from nodes #2 to #p are identical, as are those from #p to #2p. Tasks from #2 to #p

Figure 3-3 Distribution by rows

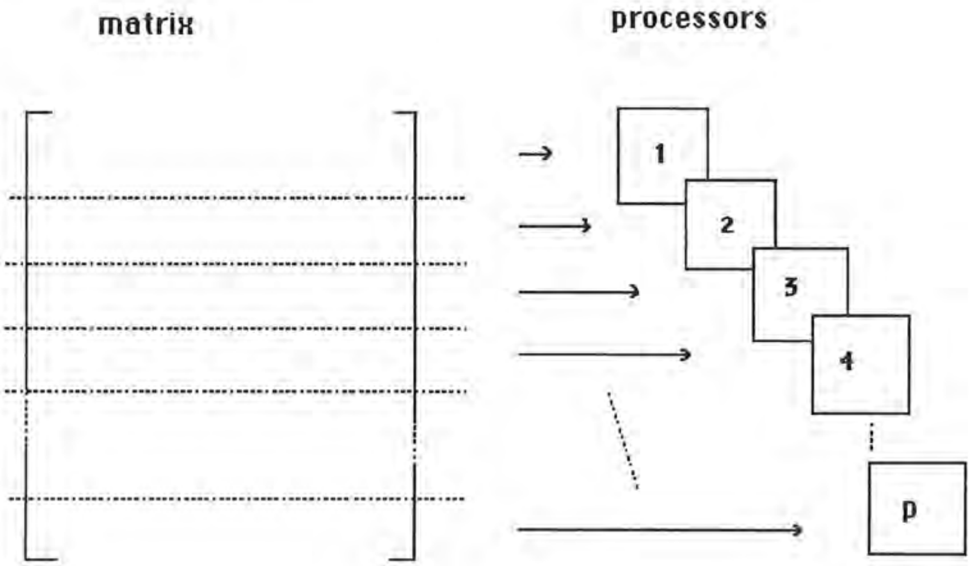
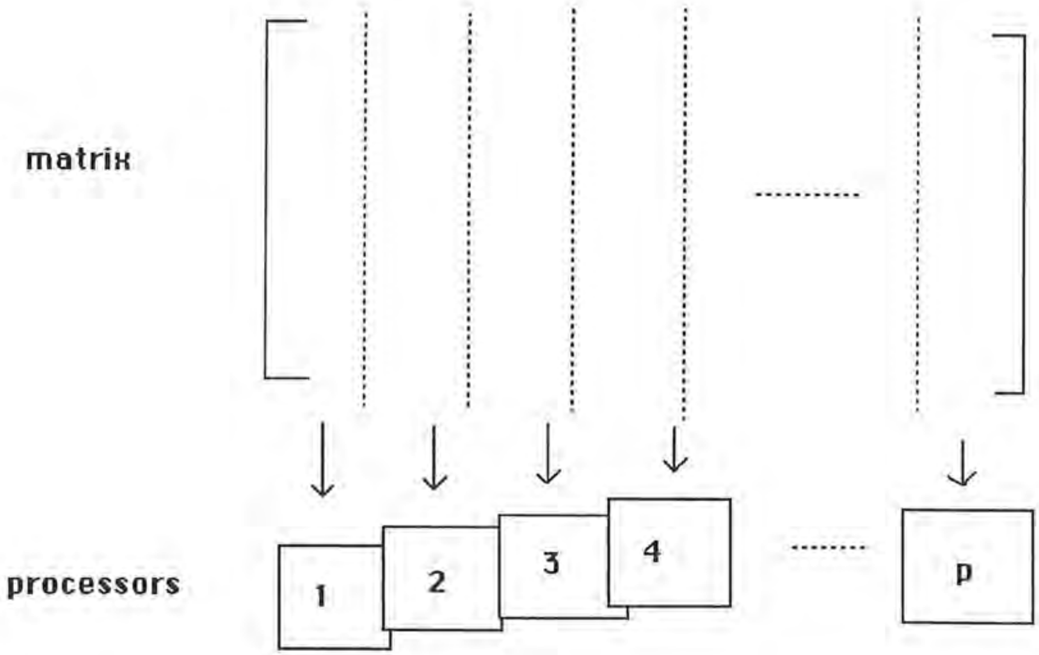


Figure 3-4 Distribution by Columns



receive the pivot column from the controller and select the local pivot row; tasks from #p to #2p select the global pivot row and send it to the controller. Then tasks #1, #2, #3, #(p-1), #p, #(p+1), #(2p-1), #2p, #(2p+1), #(3p+2) and #(4p+3) become the critical path. In this experiment, external input/output is done by tasks #1 and #(4p+3).

Theoretical execution time for one iteration is computed as follows: Communication delay, the tasks from #2 to #2p,

$$\begin{aligned} D(p, m, n) &= (P \times (\text{data size} \times (n + 3)))\ \text{communication time per byte} \\ &= \vartheta(n \times p) \end{aligned}$$

and the execution time, except for communication delay, as

$$\begin{aligned} t(p,m,n) &= (\lceil m / p \rceil\ \text{divisions}) \\ &\qquad + ((\lceil m / p \rceil \times n)\ \text{multiplications \& additions}) \\ &= \vartheta(\lceil m / p \rceil \times n) \end{aligned}$$

from the tasks #2, #2p, #2p+1, and #3p+2. Then the total execution time is $T_p(p,m,n) = D(p, m, n) + t(p,m,n) = \vartheta(n \times (p + \lceil m / p \rceil))$.

In contrast, for the serial method

$$\begin{aligned} T_s(m,n) &= (n + m - 2)\ \text{comparisons} + (m\ \text{divisions}) \\ &\qquad + (m \times n)\ \text{multiplications \& additions} \\ &= \vartheta(m \times n). \end{aligned}$$

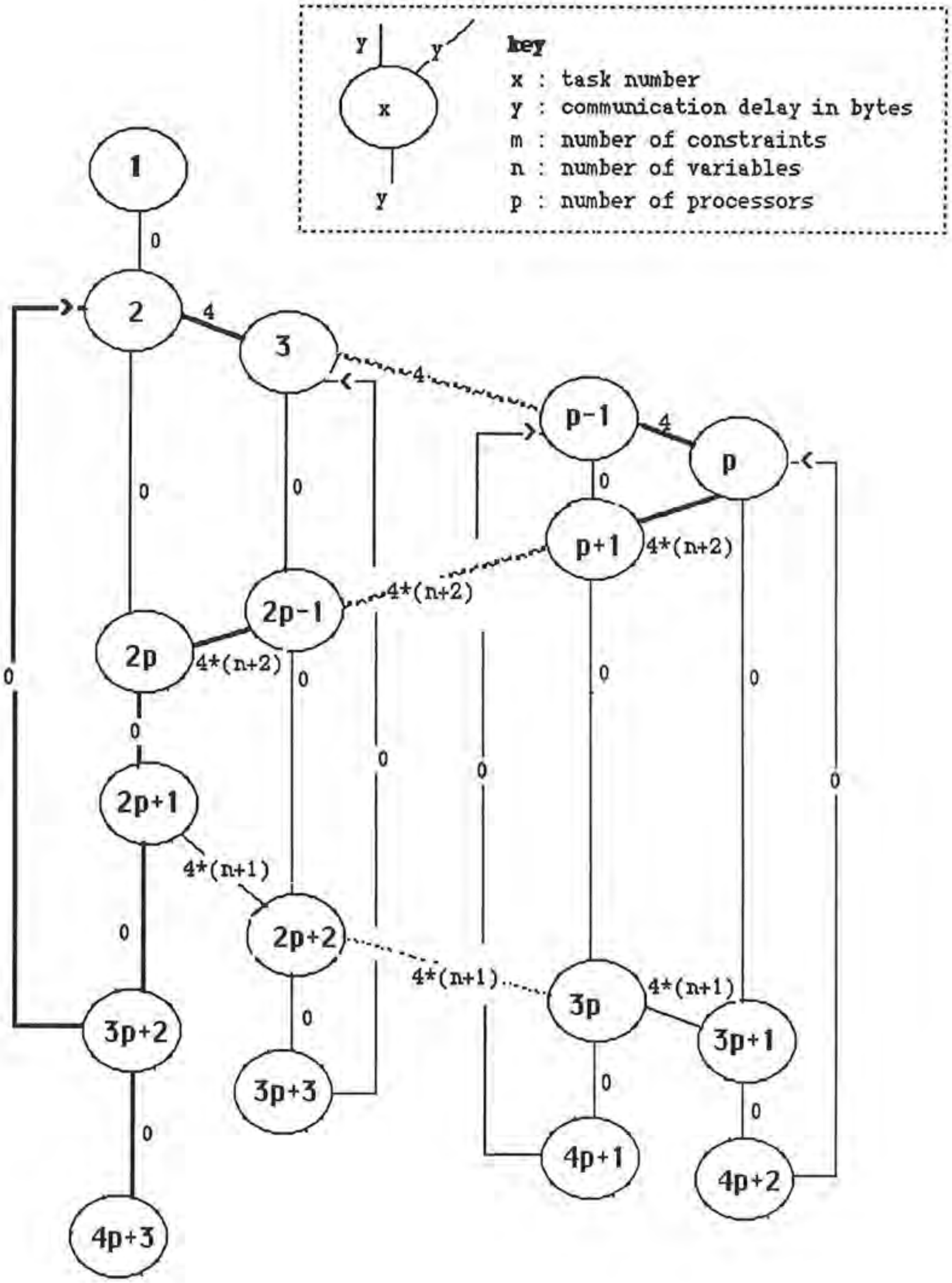
A speedup may be obtained as

$$S(p, m, n) = T_s(m,n) / T_p(p,m,n)$$

and efficiency as

$$E(p,m,n) = S(p, m, n) / p.$$

Figure 3-5 **Task Graph of Simplex Method**
with Sequential Data Loading Path



Task Description

1	Initialize the variables and find pivot column.
2	Send pivot column index to the next.
3..(p-1)	Receive pivot column index from the previous task and send it to the next.
p	Receive pivot column index. Find pivot row among its own data. Send its pivot row information to the next.
(p+1)..(2p-1)	Find pivot row among its own data. Receive pivot row information from the previous and compare its pivot row with the previous and find the smaller. Send the smaller pivot row information to the next.
2p	Find pivot row among its own data. Receive pivot row information from the previous and find the global pivot row
2p+1	Send global pivot row information.
(2p+2)..3p	Receive and send global pivot row information.
3p+1	Receive global pivot row information.
3p+2	Check the optimality of the answer. Compute new matrix. Find new pivot column index.
(3p+2)..(4p+2)	Compute new matrix.
4p+3	Terminate.

From the above, it may be seen that the execution time, t(p,m,n) varies by the factor of $\lceil m / p \rceil \times n$, but that communication delay, D(p,m,n) is dependent on the factor of $n \times p$. If the problem gets larger, when compared to the number of processors, **p**, the communication delay, **D**, can be negligeable. If the tree structured Transputer network is adapted, the communication delay is dependent on the factor of $\mathbf{n} \times log_3\mathbf{p}$. The reason why almost linear performance improvement is obtained is that the communication delay is relatively small, compared to the entire computation. Real experiments indicate the communication delay constitutes 2% to 5% of the entire execution time.

3.2.2. Distribution by columns

If the computation and data are distributed by columns, there will be similar results. The communication delay in one iteration will be

$$D\ (p, m, n) = (m\ \text{divisions}) + (P \times\ (\text{data size} \times (m + 2))(\text{communication time per byte})$$

The reason why m-divisions are involved in the communication delay is that the other processors must wait until the processor which has the pivot column selects the exiting pivot row. Also another communication delay is caused by redistribution at the beginning of phase 2 of the Simplex method. In phase 1 we build an auxiliary problem to get an initial basic problem, thus the actual problem size is $m \times (m + n)$. However in phase 2, the problem size reduces to m x n, but there will be $(m \times n)$ more communication time.

The execution time, except for communication delay is

$$t(p,m,n) = ((\lceil n / p \rceil + m + p-3) \text{ comparisons})$$

$$+ (m \text{ divisions}) + ((m \times \lceil n / p \rceil) \text{ multiplications \& additions}).$$

From the above results, it is obvious that distributing the data by rows gives better performance.

3.3 Test Cases

In testing the distribution-by-rows parallelized Simplex algorithm, 64 bit real number computation was performed with 10^{-15} precision on five 100 × 200 test cases with 99% data density and one 300 × 400 test case with 1% data density. The same speedup and efficiency were obtained for all test cases because the parallelized Simplex algorithm can not find the optimal pivot order and concentrates on the distribution of computation and the data of Simplex method step(3). So the results may be generalized because the distribution-by-rows parallelized Simplex algorithm is the same as the serial Simplex algorithm except for the distribution of computation and data to the multiprocessors. The results show an almost linear performance improvement, which exceeds the theoretical estimate of speedup.

Fig. 3-6 provides data from a Simplex problem with 100 constraints and 200 variables and Fig. 3-7 shows the case of 300 constraints and 500 variables. The only difference between the two cases is that the speedup curve is almost linear until the point for 10 processors is reached in Fig. 3-7, whereas in Fig. 3-6 the comparable number of processors is 5. The main reason for this difference is that as the data size allocated to one

Figure 3-6 Execution Time vs. Speedup

- 100 constraints and 200 variables test case
- $T(p,m,n) = 9/200 \times n \times \lceil m/p \rceil + 30$

where $m = 100$

$n = 200$

$p = 1..16$

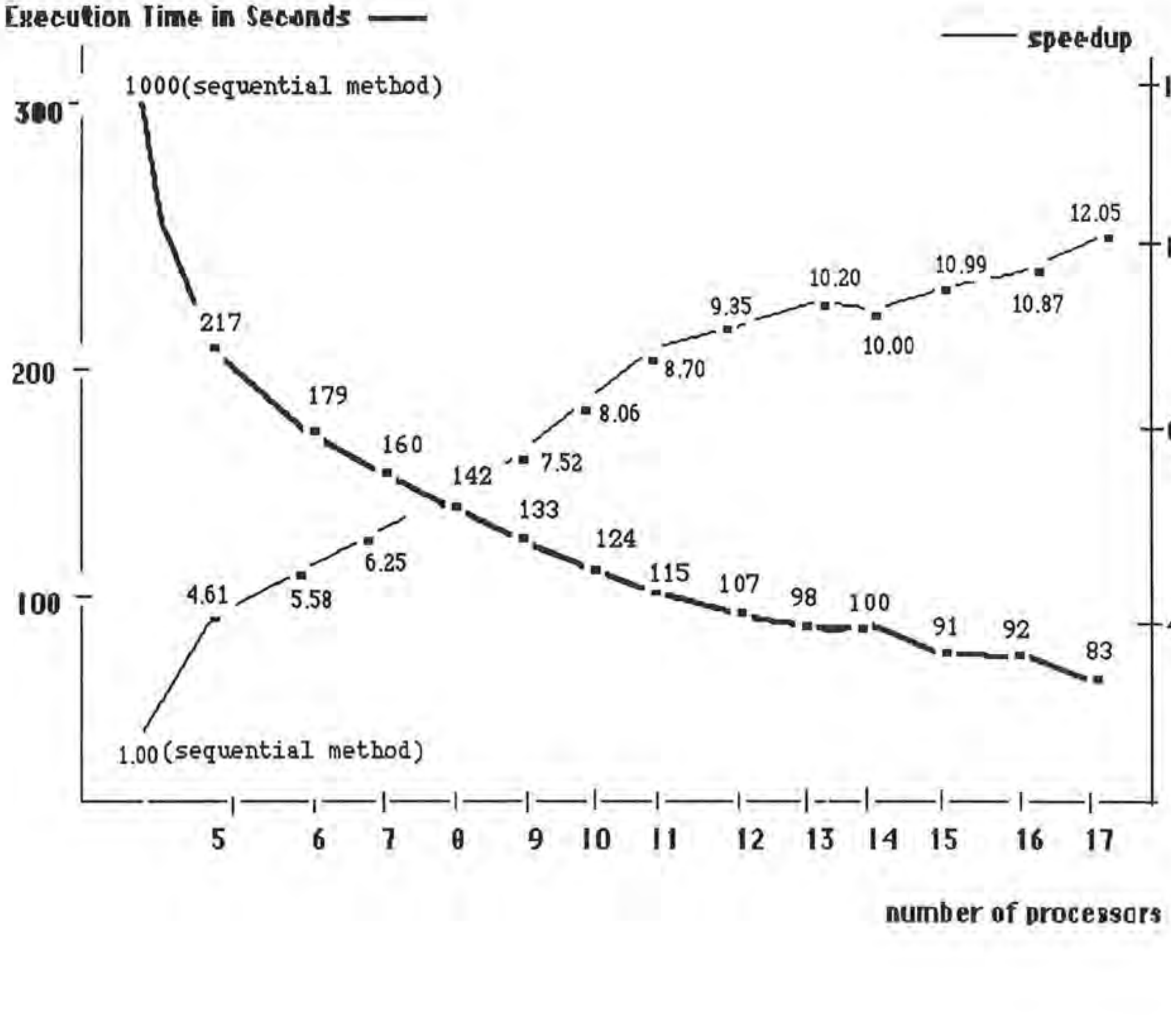


Figure 3-7

Execution Timevs. Number of Processors

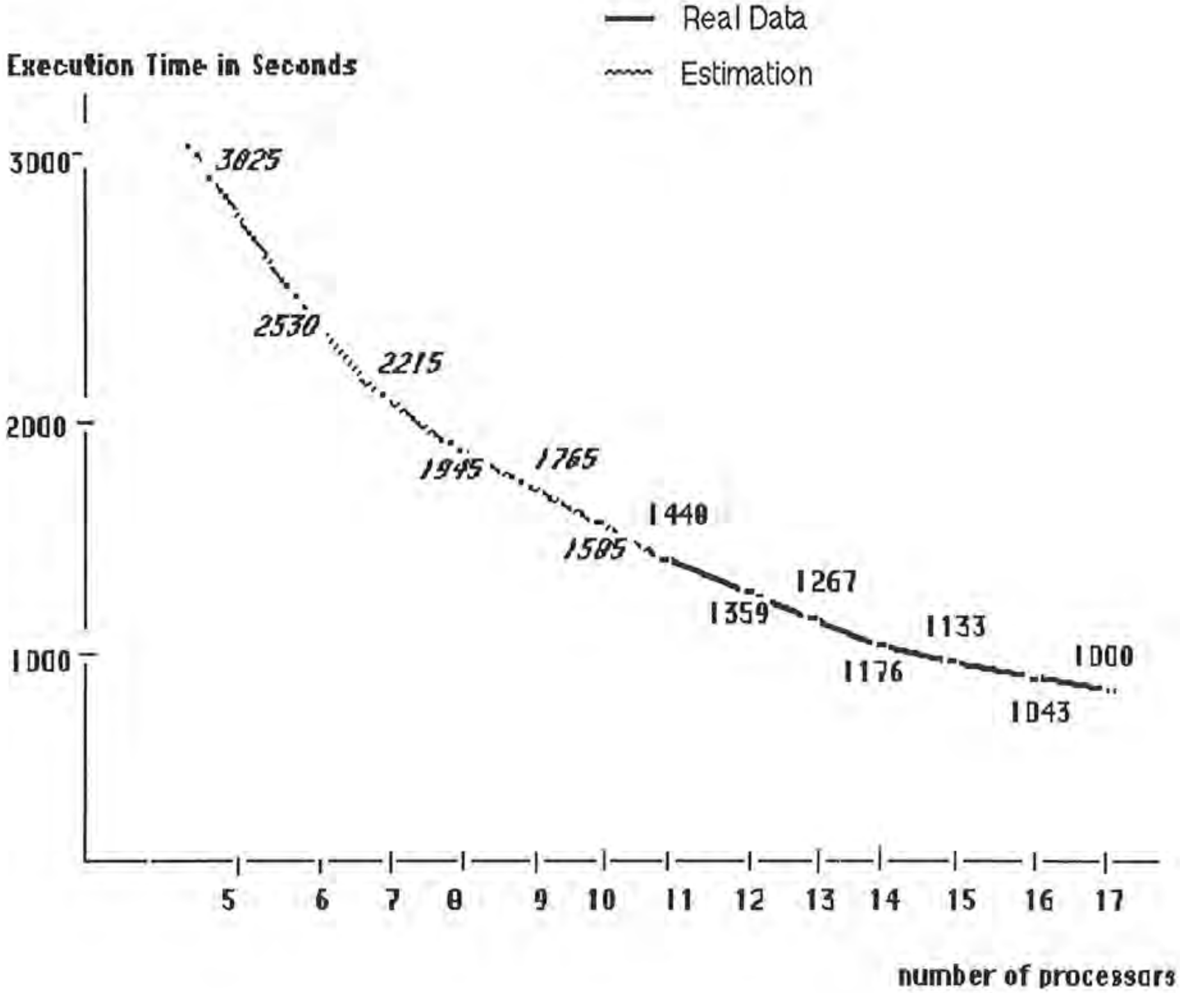
- 300 constraints and 500 varaibles test case

- $T(p,m,n) = 9 / 100 \times n \times \lceil m / p \rceil + 145$

where $m = 300$

$n = 500$

$p = 1..16$



processor gets smaller, the uneven distribution of rows to each processor can more easily affect the whole execution time. The graph which shows the linear relation between the number of rows assigned to one processor and the execution time is shown in Fig. 3-8. Execution time is affected by processors which have more rows than other processors. That is, speedup T(p, m, n), is a function of $\lceil m / p \rceil$. As the data size assigned to one processor becomes smaller, the computation time for one more row can significantly affect the whole execution time. An attempt was made to distribute the rows evenly. However if there exists a remainder row, for example m = 19 and p = 9, then each processor holds 2 rows with one row left only, the host processor was assigned first, in order to parallelize the effort of computing this redundant row and the data communication. That is, because the communication data path is sequential, the host processor is idle during the communication time from node #3 to #2p-1 in Fig. 3-5. This idle time can be used for computation of the remainer row by assigning it to the host processors first. Actually the effect is the same as reducing the communication delay.

The most interesting thing in this study is that the execution time is not dependent on communication delay which is the most critical aspect of a message passing machine. In Fig. 3-9, overhead due to parallelization did not significantly increase as the number of processors increased. This is because the Transputer system provides fine grain parallelism by fast context swithching as well as large grain parallelism. By this it is meant that even if the system under study has a sequential data communication path, the actual communication delay is two thirds of

Figure 3-8

Execution Time

vs.

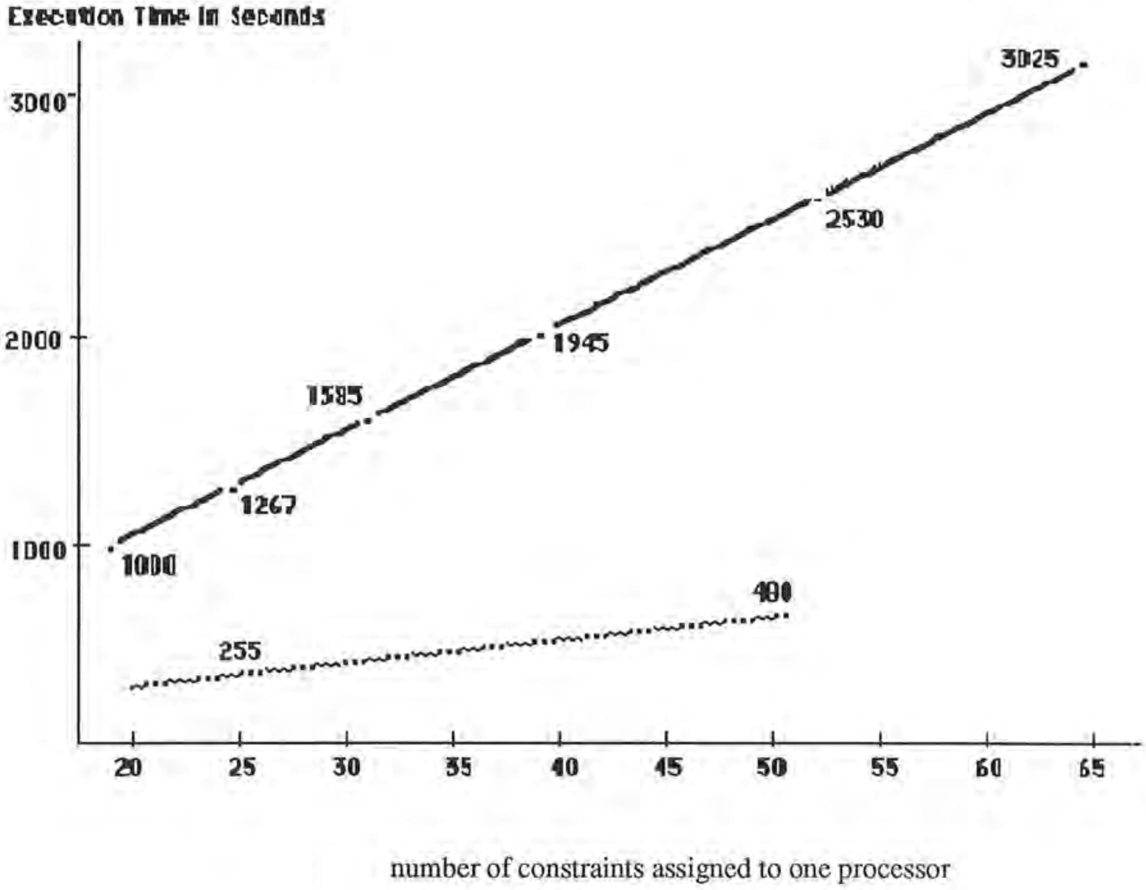
Number of Rows assigned to One Processors

___ 300 constraints and 500 varaibles test case

$$T\ (p,m,n) = 9 / 100 \times n \times \lceil m / p \rceil + 145$$

~~~~ 100 constraints and 200 variables test case

$$T\ (p,m,n) = 9 / 200 \times n \times \lceil m / p \rceil + 30$$


~~~~

D(p,m,n), shown in Fig. 3-9. As a result, the Simplex algorithm communication delay is not so significant if the problem size is big, ie., the size of the total system memory. For example in Fig. 3-9, the portion of the communication delay in the total execution time is less than 5% when the number of processors $p = 4$, and efficiency, E(p,m,n) is 95%. Actually pure communication delay is less than 3%, another 2% is due to processes that can not be parallelized. The main factor is that execution time increases for computing each additional row. The total execution time can be formulated from the test data as $T(p,m,n) = 9 / 200 \times n \times \lceil m / p \rceil + 30$ when $m = 100$, $n= 200$. The overhead due to parallelization seems to be constant compared with the whole execution time.

Every test case indicates algorithmic efficiency of 75% to 95%. This is far more than the predicted efficiency 53% to 93% (Fig. 3-10) which can be computed from section 3.2.1 and Table 3-1.

$$\begin{aligned}
D(p,100,200) &= p \times (8 \times (200 + 3)) \times (2.5 \times 10^{-6}) \\
&= 4120 \times p \times 10^{-6} \\
t(p,100,200) &= \lceil 100 / p \rceil \times (55.750 \times 10^{-6}) \\
&\qquad + (\lceil 100 / p \rceil \times 200)) \times (66.050 \times 10^{-6}) \\
&= 13265.75 \times \lceil 100 / p \rceil \times 10^{-6} \\
&\approx (1326575 / p) \times 10^{-6} \\
T_p(p,100,200) &= (4120 \times p + 13265.75 \times \lceil 100 / p \rceil) \times 10^{-6} \\
&\approx (4120 \times p + 1326575 / p) \times 10^{-6} \\
T_s(100,200) &= 1326575 \times 10^{-6} \\
S(p,100,200) &= 1326575 / (4120 \times p + 1326575 / p)
\end{aligned}$$

Figure 3-9

Execution Time

vs.

Overhead due to Parallelization

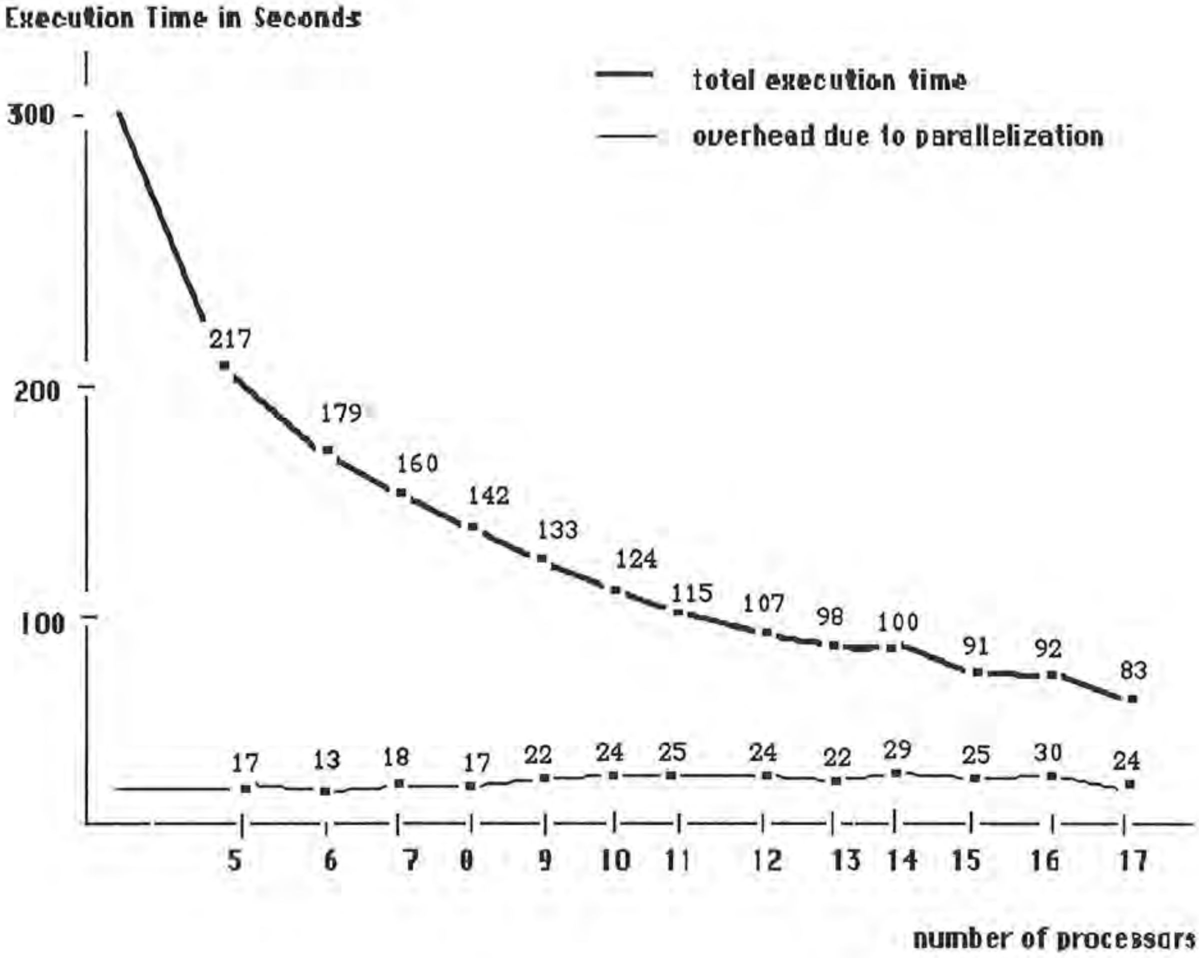
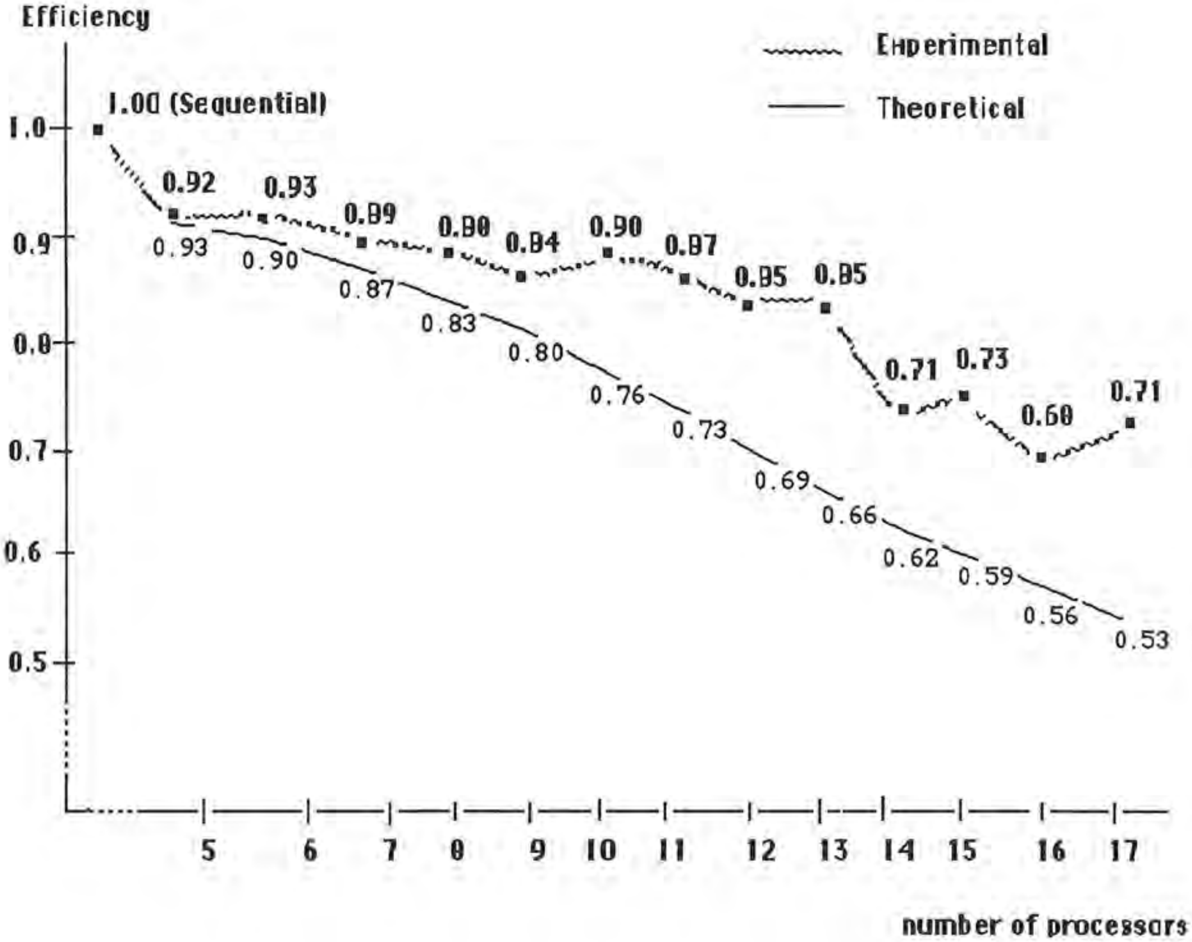


Figure 3-10 Theoretical vs. Experimental Efficiency

- 100 constraints and 200 variables case



E(p,100.200) = S(p,100,200) / p

then E(4,100,200) = 0.95 and E(16,100,200) = 0.55.

Usually, in message passing machines such as the Transputer, communication delays are the major overhead. The results of this study indicate no significant communication overhead problem, which may be attributed to a careful distribution of computation and data.

Table 3-1 Performace of operations in T414

1) Floating point operations

Real64	typical	worst
+, −	28.050	35.000
×	38.000	47.000
/	55.750	71.000

unit : micro seconds

2) Communication Speed

400 Kbytes/sec in each direction

4.0 Summary

This study presented an implementation of the Simplex method on bith mesh- and tree-structured Transputer networks with a diameters 2p and $2log_3P$ respectively. Although difficult to implement on a Transputer network, careful design of the algorithm gives nearly identical results for both mesh- and tree-structured networks. It was found that T(p,m,n) is linearly dependent on the $\lceil m / p \rceil$. In particular the experiment shows that $T(p,100,200) = \lceil 100 / p \rceil \times 9 + 30$ in the 100 by 200 case and that $T(p,300,500) = \lceil 300 / p \rceil \times 45 + 145$ in the 300 by 500 case. This is far better than earlier results for other message passing machines and almost identical to results for one shared memory machine. For example, the same algorithm implemented by Chandrashekhar Bhide, using Lynx on Crystal [3], yields an efficiency seldom above 0.5 (Table 4-1) and test cases implemented by Wu using the shared memory Sequent [12] shows an efficiency of 0.92 to 0.99 (Table 4-2). In this study, the test case indicates an efficiency of 0.89 to 0.95, whereas Wu's results show an efficiency of 0.92 to 0.95, when the number of processors is 4 to 8. By careful distribution of computation and data and fine grain parallelism, almost linear speedup pattern is obtained in this study. These results suggest that one of the most important factors in parallel programming is a good match between algorithm and architecture.

Table 4-1 **Test data by Chandrashekhar Bhide, using Lynx on Crystal** (when p = 2)

m	n	speedup	efficiency
3	6	.03	.015
5	10	.08	.040
10	20	.34	.170
15	30	.65	.325
20	40	.69	.345
25	50	.85	.425
30	60	1.00	.500
35	70	.84	.420
40	80	1.11	.555
45	90	1.01	.505
48	96	.92	.460

where m = number of constraints
n = number of variables

Table 4-2 Test data by Youfeng Wu, using Pascal on Sequent Balance

#of processors	Speedup	efficiency
1	0.994	0.994
2	1.956	0.978
3	2.904	0.968
4	3.833	0.958
5	4.741	0.948
6	5.680	0.943
7	6.511	0.930
8	7.437	0.929

References

[1]. Sayed Atef Banawan, "An Evaluation of Load Sharing In Locally Distributed Systems," Ph.D Dissertation, University of Washington, Seattle, 1987.

[2]. Donald M. Simmons, Linear Programming for Operations Research, pp. 89-162, Holden-Day Inc., 1972.

[3]. Leah H. Jamieson, Dennis B. Gannon and Robert J. Douglass, The Characteristics of Parallel Algorithms, pp. 21-63, The MIT Press, 1987.

[4]. Philip Arne Nelson, "Parallel Programming Paradigms," Ph.D. Dissertation, University of Washington, Seattle, 1987.

[5]. Transputer Reference Manual Inmos Inc., 1987.

[6]. Dick Pountain, A Tutorial Introduction to Occam programming, Inmos Inc., 1986.

[7]. Shreekant S. Thakkar, "Parallel Programming Issues and Questions," IEEE Software, 5(1):8-9, Jan 1988.

[8]. Marta Kallstrom and Shreekant S. Thakkar, "Programming Three

Parallel Computer," IEEE Software, 5(1):11-22, Jan 1988.

[9]. Boontee Kruatrachue and Ted Lewis, "Grain Size Determination for Parallel Processing," IEEE Software, 5(1):23-32, Jan 1988.

[10]. Frederick S. Hiller and Gerald J. Lieberman, Introduction to Operations Research, Holden-Day Inc, 1980.

[11]. Katta G. Murty. Linear and Combinatorial programming, Wiley and Sons Inc., 1976.

[12]. Youfeng Wu and Ted Lewis, "Performance of Parallel Simplex Algorithms," Unpublished, Oregon State University, Oregon, 1988.

Appendix A : Program Source Listings

A.1 Main Program

```
PROC simPAR2 (CHAN OF ANY keyboard, screen,
              [4]CHAN OF ANY from.user.filer, to.user.filer)
  -- *************************************** --
  -- Simplex Method Main partitioned by Rows  --
  --     By Sungwoon Choi in Oct. 19, 1987     --
  -- *************************************** --
  #USE "const2.tsr"
  #USE "userio.tsr"
  #USE "interf.tsr"
  -- vars
  [max.link]CHAN OF ANY chan.in, chan.out:
  CHAN OF ANY chan.read:

  [max.main.module.size][max.col]REAL64 in.buf:
  [max.col]REAL64 cost.buf:
  [max.row]REAL64 solution:
  [max.row]INT basis.index:
  INT pivot.row, pivot.col, row.size, col.size, main.module.size, status:
  BOOL max.problem:

  TIMER clock:
  INT time.start, time.end, run.time, input.time:

  -- channel definition
  PLACE chan.out AT link0out:
  PLACE chan.in  AT link0in:

  -- PROC input.data
  PROC input.data(CHAN OF ANY from.stream, to.stream,
                    chan.read, VAL INT fold.no, INT status)
    INT data.size, row.size, col.size, file.error:
    SEQ
      write.full.string (screen, "%% File read ...*c*n")
      -- channel declarations
      CHAN OF INT filekeys:
      CHAN OF INT keyboard IS filekeys: -- channel from simulated keyboard
      CHAN OF ANY echo:
      -- echo channel with scope local to this PAR only
      CHAN OF ANY screen IS echo:

      PAR
        --------------------------------------------------
        -- read data from file and send to distributor
        -- vars
        [512]BYTE buf:
        INT kchar, row, col, len, index:
        REAL64 in.data:
        BOOL max.problem, error:

        SEQ
          -- initialize index
          status := 0
          data.size := 0
          kchar := 0

          -- read pre data (size, max or min flag)
          read.char (keyboard, kchar)

          -- determine whether this is max or min problem
          read.char (keyboard, kchar)
          WHILE (kchar = ' '(INT))
            read.char (keyboard, kchar)
          read.char (keyboard, kchar)
          IF
          -- determine whether this is the max problem or min problem
```

```
      (kchar = 'I'(INT)) OR (kchar = 'i'(INT))
        max.problem := FALSE
      TRUE
        max.problem := TRUE
    read.char (keyboard, kchar)
    read.char (keyboard, kchar)

    -- read row and column size
    read.echo.int (keyboard, screen, row.size, kchar)
    read.echo.int (keyboard, screen, col.size, kchar)

    -- read and send real data
    IF
      (row.size > max.row) OR (col.size > max.col)
        SEQ -- error, data size is too big
          chan.read ! ft.error
          status := 4
      (row.size < max.cpu)
        SEQ -- error, data size is too small
          chan.read ! ft.error
          status := 5
      TRUE
        -- read and send data
        INT char:
        SEQ
          chan.read ! kchar; max.problem; row.size; col.size

          read.char (keyboard, kchar)
          WHILE (kchar <> '%'(INT)) AND (kchar <> ft.terminated)
            SEQ
              data.size := data.size + 1
              read.echo.int(keyboard, screen, row, kchar)
              read.echo.int(keyboard, screen, col, kchar)
              read.echo.real64(keyboard, screen, in.data, kchar)
              chan.read ! kchar; row; col; in.data
              read.char (keyboard, kchar)
          chan.read ! ft.terminated

    -- write end stream
    IF
      (kchar >= 0) OR (kchar = ft.number.error)
        keystream.sink (keyboard)
        -- consume rest of the keyboard file
      TRUE
        SKIP
    write.endstream (screen) -- terminate scrstream.sink

  --------------------------------------------------
  -- mux
  keystream.from.file (from.stream, to.stream,
                       keyboard, fold.no, file.error)

  --------------------------------------------------
  -- consume everything echoed
  scrstream.sink (screen)    -- consume everything echoed

  --------------------------------------------------
-- test input.error, if OK tabulate
IF
  (status = 0) AND (file.error = 0)
    SEQ
      write.full.string (screen, "%% File read OK    : ")
      write.int (screen, row.size, 0)
      write.full.string (screen, " x ")
      write.int (screen, col.size, 0)
      newline (screen)
      write.full.string (screen, "%% Real Data Size  : ")
      write.int (screen, data.size, 0)
      newline (screen)
  file.error <> 0
    status := file.error
  TRUE
    SKIP
```

```
:

-- PROC distributeData
PROC distribute.data([]CHAN OF ANY chan.in, chan.out, CHAN OF ANY chan.read,
      [][]REAL64 in.buf, []REAL64 cost.buf, []INT basis.index,
      INT row.size, col.size, main.module, BOOL max.problem)
  -- vars
  INT kchar, row, col, module.size, modular:
  REAL64 in.data:

  SEQ
    -- initialize in.buf for main module
    SEQ row = 0 FOR max.main.module.size
      SEQ col = 0 FOR max.col
        in.buf[row][col] := 0.0(REAL64)

    SEQ col = 0 FOR max.col
      cost.buf[col] := 0.0(REAL64)

    SEQ row = 0 FOR max.row
      basis.index[row] := 0

    chan.read ? kchar
    IF
      kchar = ft.error
        SKIP
      TRUE
        SEQ
          -- receive problem parameters
          chan.read ? max.problem; row.size; col.size

          -- compute module.size and send size and constraint condition
          module.size := row.size / (max.cpu PLUS 1)
          modular := row.size MINUS (module.size TIMES (max.cpu PLUS 1))
          IF
            modular > 0
              SEQ
                main.module := module.size PLUS 1
                modular := modular MINUS 1
            TRUE
              main.module := module.size
          chan.out[chan.num] ! main.module; module.size; modular

          -- receive and distribute data
          chan.read ? kchar
          chan.out[chan.num] ! kchar
          WHILE (kchar <> ft.terminated)
            SEQ
              chan.read ? row; col; in.data; kchar
              IF
                row = 0 -- save object function for phase II
                  IF
                    max.problem
                      cost.buf[col] := in.data
                    TRUE
                      cost.buf[col] := in.data
                row < main.module
                  in.buf[row][col] := in.data
                TRUE
                  chan.out[chan.num] ! row; col; in.data; kchar
              -- new cost
              IF
                (row > 0) AND (col < ((col.size MINUS row.size) PLUS 1))
                  in.buf[0][col] := in.buf[0][col] - in.data
                TRUE
                  SKIP

          -- get basis variable index
          SEQ index = 1 FOR (row.size MINUS 1)
            -- get basic variable index vector
            basis.index[index] := index PLUS (col.size MINUS row.size)
```

```
:

-- PROC computation
PROC computation ([][]REAL64 in.buf, []REAL64 pivot.row.value,
                    INT pivot.row, pivot.col, row.size, col.size)
  [max.main.module.size]REAL64 comp.base:
  SEQ
    -- compute new pivot column for computaional base
    SEQ index = 0 FOR row.size
      comp.base [index] := (-(in.buf[index][pivot.col] /
                                           pivot.row.value[pivot.col]))
    IF
      pivot.row < row.size
        comp.base [pivot.row] := (1.0(REAL64) - pivot.row.value[pivot.col])
                                  / pivot.row.value[pivot.col]
      TRUE
        SKIP

    -- compute module
    -- compute new matrix using basis column
    SEQ col = 0 FOR col.size
      SEQ row = 0 FOR row.size
        in.buf[row][col] := in.buf[row][col] +
                                    (comp.base[row] * pivot.row.value[col])

:

-- PROC find.min
PROC find.min ([]REAL64 buf, INT size, min.index)
  SEQ
    min.index := 1
    SEQ index = 1 FOR size-1
      IF
        buf[index] < buf[min.index]
          min.index := index
        TRUE
          SKIP
:

-- PROC iteration
PROC iteration ([]CHAN OF ANY chan.in, chan.out, [][]REAL64 in.buf,
   []INT basis.index, INT row.size, col.size, main.module.size, status)
  -- vars
  [max.main.module.size]REAL64 base:
  [max.col]REAL64 pivot.row.value, sub.pivot.row.value:
  REAL64 base.value, sub.base.value, epsilon:
  INT pivot.row, pivot.col, sub.pivot.row:
  BOOL degenerate:

  SEQ
    -- find pivot column
    find.min (in.buf[0], col.size, pivot.col)
    IF
      in.buf[0][0] > ZERO
        status := 2 -- no feasible solution
      in.buf[0][0] > (-ZERO)
        status := 0 -- normal end
      TRUE
        status := 1 -- more to compute
    chan.out[chan.num] ! status

    WHILE status = 1
      SEQ
        -- find pivot row
        chan.out[chan.num] ! pivot.col
        degenerate := TRUE
        epsilon := ZERO
        WHILE degenerate
          SEQ
            PAR
              -- find current processor's pivot row
              SEQ
                pivot.row := 1
```

```
                SEQ row = 1 FOR (main.module.size MINUS 1)
                  SEQ
                    IF
                      (in.buf[row][pivot.col] > ZERO)
                        IF
                          in.buf[row][0] < (-ZERO)
                            base[row] := MAX.REAL64
                          TRUE
                            base[row] := in.buf[row][0]
                                           / in.buf[row][pivot.col]
                      TRUE
                        base[row] := MAX.REAL64
                    IF
                      base[row] < base[pivot.row]
                        pivot.row := row
                      TRUE
                        SKIP
                base.value := base[pivot.row]
              -- receive sub processor's pivot row
              chan.in[chan.num] ? sub.pivot.row;
                                 [sub.pivot.row.value FROM 0 FOR col.size];
                                  sub.base.value
            degenerate := FALSE
            IF
              base.value = sub.base.value
                -- degenerate case (THE LEXICO MINIMUM RATIO RULE)
                SEQ
                  degenerate := TRUE
                  chan.out[chan.num] ! degenerate
                  SEQ row = 1 FOR (main.module.size MINUS 1)
                    SEQ
                      epsilon := epsilon / 2.0(REAL64)
                      in.buf[row][0] := in.buf[row][0] + epsilon
                  chan.out[chan.num] ! epsilon

              base.value > sub.base.value
                -- sub processor's pivot row is global pivot row
                SEQ
                  chan.out[chan.num] ! degenerate
                  pivot.row := sub.pivot.row
                  [pivot.row.value FROM 0 FOR col.size] :=
                                [sub.pivot.row.value FROM 0 FOR col.size]
                  base.value := sub.base.value

              TRUE
                -- own pivot row is global pivot row
                SEQ
                  chan.out[chan.num] ! degenerate
                  [pivot.row.value FROM 0 FOR col.size] :=
                               [in.buf[pivot.row] FROM 0 FOR col.size]

        chan.out[chan.num] ! pivot.row;
                             [pivot.row.value FROM 0 FOR col.size]

        IF
          base.value >= MAX.REAL64
            status := 3 -- unboundness
          TRUE
            SEQ
              computation (in.buf, pivot.row.value, pivot.row, pivot.col,
                                 main.module.size, col.size)
              -- find pivot column
              basis.index[pivot.row] := pivot.col
              find.min (in.buf[0], col.size, pivot.col)
              IF
                in.buf[0][pivot.col] >= (-ZERO)
                  status := 0 -- end
                TRUE
                  SKIP

        chan.out[chan.num] ! status
:
```

```
-- PROC build.new.object
PROC build.new.object ([]CHAN OF ANY chan.in, chan.out, [][]REAL64 in.buf,
                    []REAL64 cost.buf, []INT basis.index, BOOL max.problem,
                       INT row.size, col.size, main.module.size)
  REAL64 sum:
  INT size, real.col.size, kchar:
  [max.col]BOOL basic:
  [max.col]REAL64 pivot.row.value:
  SEQ
    -- set the basic variable indicator
    SEQ index = 0 FOR col.size
      basic[index] := FALSE
    SEQ index = 1 FOR row.size-1
      basic[basis.index[index]] := TRUE

    -- when the artificial variable is in the basis
    SEQ
      kchar := 0
      real.col.size := (col.size MINUS row.size) PLUS 1
      SEQ row = 1 FOR (row.size MINUS 1)
        IF
        -- then this is artificial var
          (basis.index[row] >= real.col.size)
            SEQ
              -- process pivot row information
              pivot.row := row
              chan.out[chan.num] ! kchar; pivot.row
              chan.in[chan.num] ? [pivot.row.value FROM 0 FOR col.size]
              IF
                pivot.row < main.module.size
                  pivot.row.value :=[in.buf[pivot.row] FROM 0 FOR col.size]
                TRUE
                  SKIP
              IF
                (pivot.row.value[0]>(-ZERO)) AND (pivot.row.value[0]< ZERO)
                  -- select pivot column and zero to zero pivot
                  SEQ
                    pivot.col := 1
                    WHILE (pivot.col < real.col.size) AND
                          (basic[pivot.col] OR
                           ((pivot.row.value[pivot.col] > (-ZERO)) AND
                              (pivot.row.value[pivot.col] < ZERO)))
                      pivot.col := pivot.col + 1
                    IF
                      pivot.col = real.col.size
                        -- redundant case
                        PAR
                          chan.out[chan.num] ! TRUE
                          SEQ
                            write.full.string(screen,"%% Redundant Case: ")
                            write.int(screen, row, 0)
                            newline(screen)

                      TRUE
                        SEQ
                          PAR
                            computation (in.buf, pivot.row.value,
                                              pivot.row, pivot.col,
                                                main.module.size, col.size)
                            chan.out[chan.num] ! FALSE;
                                     [pivot.row.value FROM 0 FOR col.size];
                                     pivot.col
                          basic[pivot.col] := TRUE
                          basis.index[pivot.row] := pivot.col

                TRUE
                  -- send SKIP message to subs
                  PAR
                    chan.out[chan.num] ! TRUE
                    SEQ
                      write.full.string(screen, "%% Redundant Case ! : ")
                      write.int(screen, row, 0)
                      newline(screen)
```

```
          TRUE
            SKIP
      chan.out[chan.num] ! ft.terminated

    -- send real col size
    col.size := real.col.size
    chan.out[chan.num] ! col.size

    -- send new cost coeficient
    size := row.size MINUS main.module.size
    chan.out[chan.num] ! [basic FROM 0 FOR col.size]; size
    SEQ row = main.module.size FOR size
      chan.out[chan.num] ! cost.buf[basis.index[row]]

    -- compute new object function
    SEQ col = 0 FOR col.size
      SEQ
        IF
          basic[col] = FALSE
            SEQ
              PAR
                SEQ
                  in.buf[0][col] := 0.0 (REAL64)
                  SEQ row = 1 FOR main.module.size-1
                    in.buf[0][col] := in.buf[0][col] +
                          (in.buf[row][col] * cost.buf[basis.index[row]])
                chan.in[chan.num] ? sum
              in.buf[0][col] := (in.buf[0][col] + sum) - cost.buf[col]
          TRUE
            SKIP

:

-- PROC output.result
PROC output.result (CHAN OF ANY from.stream, to.stream,
          []CHAN OF ANY chan.in, [max.row]REAL64 solution, []INT basis.index,
                          INT input.time, run.time, row.size, col.size)
  SEQ
    -- PROC recieve.soultion
    PROC receive.solution ([]CHAN OF ANY chan.in, []REAL64 solution)
      INT kchar, row:
      SEQ
        chan.in[chan.num] ? kchar
        WHILE kchar <> ft.terminated
          chan.in[chan.num] ? row; solution[row]; kchar
    :

    -- PROC writings
    PROC writings (CHAN OF ANY screen, []REAL64 solution,
           []INT basis.index, INT input.time, run.time, row.size, col.size)
      SEQ
        write.full.string(screen, "##  Simplex Method (cpu=")
        write.int(screen, max.cpu, 0)
        write.full.string(screen, ",REAL64)   Start  ##*c*n")
        write.full.string(screen, "##  Problem Size  : ")
        write.int(screen, row.size, 0)
        write.full.string(screen, " x ")
        write.int(screen, col.size, 0)
        newline(screen)
        write.full.string(screen, "##  Input Time    : ")
        write.int(screen, input.time, 0)
        newline(screen)
        write.full.string(screen, "##  R u n Time    : ")
        write.int(screen, run.time, 0)
        newline(screen)
        write.full.string(screen, "##  Optimal value : ")
        write.real64(screen, solution[0], 0, 2)
        newline(screen)
        write.full.string(screen, "##  Simplex.method (cpu=")
        write.int(screen, max.cpu, 0)
        write.full.string(screen, ",REAL64)   E n d  ##*c*n")
        newline(screen)
```

```
        write.full.string(screen, "##  Solution Start*c*n")
        SEQ row = 0 FOR row.size
          SEQ
            write.int(screen, basis.index[row], 6)
            write.real64(screen, solution[row], 8, 0)
            newline(screen)
        write.full.string(screen, "##  Solution E n d*c*n")
    :

    -- vars
    CHAN OF ANY fromprog, tofile:

    PAR
      -----------------------------------
      -- data writing
      SEQ
        receive.solution(chan.in, solution)
        writings(tofile,solution,basis.index,input.time,run.time, row.size,
                   col.size)
        write.endstream(tofile)

      -----------------------------------
      -- COMMENT screen echo (optional)

      -----------------------------------
      -- mux
      INT result, fold.no:
      SEQ
        fold.no := 0
        scrstream.to.file (tofile, from.stream,
                               to.stream, "output", fold.no, result)
        IF
          result = 0
            SKIP
          TRUE
            STOP -- only alternative is to call scrstream.sink(tofile)

      -----------------------------------
    -- press any to continue
    write.full.string(screen, "Press [ANY] key to continue")
    INT any:
    read.char(keyboard , any)

:

-- PROC error.message
PROC error.message (CHAN OF ANY keyboard, screen, INT status)
  -- error code explanation
  -- *************************************** --
  --  status   0 : normal fin                --
  --           1 : now doing                 --
  --           2 : no feasible solution      --
  --           3 : unboundness               --
  --           4 : data size is too big      --
  --           5 : data size is too small    --
  --      others : system error number       --
  -- *************************************** --

  SEQ
    IF
      (status = 0) OR (status = 1)
        SKIP
      status = 2
        write.full.string (screen, "%% ERROR : NO feasible solution !")
      status = 3
        write.full.string (screen, "%% ERROR : Unboundness !")
      status = 4
        write.full.string (screen, "%% ERROR : Data Size is Too Big !")
      status = 5
        write.full.string (screen, "%% ERROR : Data Size is Too small !")
      TRUE
        SEQ
          write.full.string (screen, "%% File read error : ")
```

```
            write.int (screen, status, 0)
      -- press any to continue
      newline (screen)
      write.full.string(screen, "Press [ANY] key to continue")
      INT any:
      read.char(keyboard , any)
      newline(screen)

  :

  SEQ
    -- main procedure
    -- input data
    clock ? time.start
    PAR
      input.data (from.user.filer[2], to.user.filer[2],
                                            chan.read, 3, status)
      distribute.data(chan.in, chan.out, chan.read,
                          in.buf, cost.buf, basis.index,
                          row.size, col.size, main.module.size, max.problem)
    clock ? time.end
    input.time := time.end MINUS time.start

    IF
      status = 0
        -- phase I
        SEQ
          write.full.string(screen,"%% SImplex Method Start(REAL64) ...*c*n")
          clock ? time.start
          chan.out[chan.num] ! col.size
          iteration (chan.in, chan.out,in.buf, basis.index, row.size,
                         col.size, main.module.size, status)
          -- Feasibility Check
          IF
            (in.buf[0][0] > ZERO) OR (in.buf[0][0] < (-ZERO))
              status := 2 -- infeasible solution
            TRUE
              SKIP

          IF
            status = 0
              -- phase II
              SEQ
                build.new.object (chan.in,chan.out,in.buf,cost.buf,
                                    basis.index,max.problem, row.size,
                                    col.size, main.module.size)
                iteration (chan.in, chan.out,
                           in.buf, basis.index, row.size, col.size,
                            main.module.size, status)

                IF
                  status = 0
                    -- output result
                    SEQ
                      clock ? time.end
                      write.full.string (screen, "%% SImplex Method E n d
                                                   (REAL64) ...*c*n")
                      run.time := time.end MINUS time.start
                      SEQ row = 0 FOR main.module.size
                        solution[row] := in.buf[row][0]
                      output.result (from.user.filer[0], to.user.filer[0],
                                  chan.in,solution, basis.index,
                                  input.time, run.time, row.size, col.size)

                  TRUE
                    error.message (keyboard, screen, status)
            TRUE
              error.message (keyboard, screen, status)
      TRUE
        error.message (keyboard, screen, status)
:
```

A.2 Subprograms implemented on the Subprocessors

```
-- node
PROC node (VAL INT cpu, CHAN OF ANY from.root, to.root, from.sub, to.sub)
  -- ************************************************ --
  -- Simplex Method Segmentation partitioned by Rows  --
  --       By Sungwoon Choi in Oct. 19, 1987          --
  -- ************************************************ --
  #USE "const2.tsr"
  -- vars
  [max.module][max.col]REAL64 in.buf:
  [max.module]REAL64 cost.buf, base:
  [max.col]REAL64 pivot.row.value:
  [max.col]BOOL basic:
  INT row.size, col.size, pre.size, full.size, row, col, kchar, size, status:
  INT module.size, modular, pivot.row, pivot.col:
  REAL64 in.data, sum:
  BOOL redundant:

  -- PROC inverseMatrix
  PROC inverseMatrix ([][]REAL64 in.buf, []REAL64 pivot.row.value,
                      INT pivot.row, pivot.col, pre.size, row.size, col.size)
    SEQ
      full.size := pre.size PLUS row.size
      SEQ index = 0 FOR row.size
        base [index] := (-(in.buf[index][pivot.col] /
                                             pivot.row.value[pivot.col]))
      IF
        (pivot.row >= pre.size) AND (pivot.row < full.size)
          base [pivot.row MINUS pre.size] := (1.0(REAL64) -
                   pivot.row.value[pivot.col]) / pivot.row.value[pivot.col]
        TRUE
          SKIP

      -- compute new matrix using basis column
      SEQ col = 0 FOR col.size
        SEQ row = 0 FOR row.size
          in.buf[row][col] := in.buf[row][col] +
                                       (base[row] * pivot.row.value[col])
  :

  -- PROC computation
  PROC computation (CHAN OF ANY from.root, to.root, from.sub, to.sub,
                    [][]REAL64 in.buf, INT pre.size, row.size, col.size, status)
    -- vars
    VAL running IS 1:
    INT sub.pivot.row:
    [max.col]REAL64 sub.pivot.row.value:
    REAL64 base.value, sub.base.value, epsilon:
    BOOL degenerate:

    SEQ
      -- receive and send status
      from.root ? status
      to.sub    ! status

      WHILE status = running
        SEQ
          -- receive and send pivot column
          from.root ? pivot.col
          to.sub    ! pivot.col

          degenerate := TRUE
          WHILE degenerate
            SEQ
              PAR
                -- find current processor's pivot row
                SEQ
                  pivot.row := 0
```

```
                SEQ row = 0 FOR row.size
                  SEQ
                    IF
                      (in.buf[row][pivot.col] > ZERO)
                        IF
                          in.buf[row][0] < (-ZERO)
                            base[row] := MAX.REAL64
                          TRUE
                            base[row] := in.buf[row][0] / in.buf[row][pivot.col]
                      TRUE
                        base[row] := MAX.REAL64
                    IF
                      base[row] < base[pivot.row]
                        pivot.row := row
                      TRUE
                        SKIP
                base.value  := base[pivot.row]
                [pivot.row.value FROM 0 FOR col.size] :=
                                 [in.buf[pivot.row] FROM 0 FOR col.size]

              -- receive pivot row from sub processors
              from.sub ?  sub.pivot.row;
                          [sub.pivot.row.value FROM 0 FOR col.size];
                          sub.base.value

            -- select pivot row and send to and receive from the root
            IF
              base.value > sub.base.value
                to.root !  sub.pivot.row;
                           [sub.pivot.row.value FROM 0 FOR col.size];
                           sub.base.value
              TRUE
                to.root ! pivot.row PLUS pre.size;
                          [pivot.row.value FROM 0 FOR col.size];
                          base.value
            from.root ? degenerate
            to.sub ! degenerate
            IF
              degenerate
                -- the lexico minimum ratio rule
                SEQ
                  from.root ? epsilon
                  SEQ row = 0 FOR row.size
                    SEQ
                      epsilon := epsilon / 2.0(REAL64)
                      in.buf[row][0] := in.buf[row][0] + epsilon
                  to.sub ! epsilon

              TRUE
                SKIP

        -- compute inverse matrix
        from.root ? pivot.row; [pivot.row.value FROM 0 FOR col.size]
        PAR
          to.sub ! pivot.row; [pivot.row.value FROM 0 FOR col.size]
          inverseMatrix (in.buf, pivot.row.value,
                         pivot.row, pivot.col, pre.size, row.size, col.size)

        -- receive and send status
        from.root ? status
        to.sub    ! status

:

SEQ
  WHILE TRUE
    SEQ
      -- initial data receive
      SEQ row = 0 FOR max.module
        SEQ col = 0 FOR max.col
          in.buf[row][col] := 0.0 (REAL64)

      from.root ? pre.size; module.size; modular
```

```
-- set row.size
IF
  modular > 0
    SEQ
      row.size := module.size PLUS 1
      modular := modular MINUS 1
  TRUE
    row.size := module.size

PAR
  to.sub ! pre.size PLUS row.size ; module.size; modular
  SEQ
    from.root ? kchar
    WHILE (kchar <> ft.terminated)
      SEQ
        from.root ? row; col; in.data
        IF
          row < (pre.size PLUS row.size)
            in.buf[row MINUS pre.size][col] := in.data
          TRUE
            to.sub ! kchar; row; col; in.data
        from.root ? kchar
    to.sub ! kchar

-- phase I
from.root ?  col.size
to.sub ! col.size
computation (from.root, to.root, from.sub, to.sub,
                          in.buf, pre.size, row.size, col.size, status)

IF
  status = 0
    SEQ
      -- when the artificial variable is in the basis
      from.root ? kchar
      to.sub ! kchar
      WHILE (kchar <> ft.terminated)
        SEQ
          from.root ? pivot.row
          to.sub ! pivot.row
          from.sub  ? [pivot.row.value FROM 0 FOR col.size]
          IF
            (pivot.row >= pre.size) AND
                       (pivot.row < (pre.size PLUS row.size))
              [pivot.row.value FROM 0 FOR col.size] :=
                     [in.buf[pivot.row MINUS pre.size] FROM 0 FOR col.size]
            TRUE
              SKIP
          to.root ! [pivot.row.value FROM 0 FOR col.size]
          -- zero to zero pivot
          SEQ
            from.root ? redundant
            to.sub ! redundant
            IF
              redundant
                SKIP
              TRUE
                SEQ
                  from.root ? [pivot.row.value FROM 0 FOR col.size];
                              pivot.col
                  PAR
                    inverseMatrix (in.buf, pivot.row.value,
                         pivot.row, pivot.col, pre.size, row.size, col.size)
                    to.sub  ! [pivot.row.value FROM 0 FOR col.size];
                              pivot.col

          from.root ? kchar
          to.sub ! kchar

      -- phase II
      SEQ
        from.root ? col.size
        to.sub ! col.size
```

```
                -- receive root processor's data and send to ths sub
                from.root ? [basic FROM 0 FOR col.size]; size
                to.sub    ! [basic FROM 0 FOR col.size]; (size MINUS row.size)

                from.root ? [cost.buf FROM 0 FOR row.size]
                REAL64 temp.data:
                SEQ index = row.size FOR (size MINUS row.size)
                  SEQ
                    from.root ? temp.data
                    to.sub ! temp.data

                -- form a new BFS
                REAL64 sum:
                SEQ index1 = 0 FOR col.size
                  IF
                    basic[index1] = FALSE
                      SEQ
                        sum := 0.0 (REAL64)
                        SEQ
                          SEQ index2 = 0 FOR row.size
                            sum := sum +
                                   (in.buf[index2][index1] * cost.buf[index2])
                          from.sub ? in.data
                        sum := in.data + sum
                        to.root ! sum
                    TRUE
                      SKIP

                computation (from.root, to.root, from.sub, to.sub,
                                    in.buf, pre.size, row.size, col.size, status)

              -- send solution
              SEQ
                kchar := 0
                SEQ row = 0 FOR row.size
                  to.root ! kchar; (row PLUS pre.size); in.buf[row][0]
                from.sub ? kchar
                WHILE kchar <> ft.terminated
                  SEQ
                    from.sub ? row; in.data
                    to.root  ! kchar; row; in.data
                    from.sub ? kchar
                to.root ! kchar

          TRUE
            SKIP
:

-- lastnode
PROC lastnode (VAL INT cpu, CHAN OF ANY from.root, to.root)
  -- *********************************************** --
  -- Simplex Method Segmentation partitioned by Rows  --
  --       By Sungwoon Choi in Oct. 19, 1987          --
  -- *********************************************** --
  #USE "const2.tsr"
  -- vars
  [max.module][max.col]REAL64 in.buf:
  [max.module]REAL64 cost.buf, base:
  [max.col]REAL64 pivot.row.value:
  [max.col]BOOL basic:
  INT row.size, col.size, pre.size, row, col, kchar, size, modular, status:
  INT pivot.row, pivot.col, full.size:
  REAL64 in.data, sum:
  BOOL redundant:

  -- PROC inverseMatrix
  PROC inverseMatrix ([][]REAL64 in.buf, []REAL64 pivot.row.value,
                      INT pivot.row, pivot.col, pre.size, row.size, col.size)
    SEQ
      full.size := pre.size PLUS row.size
      SEQ index = 0 FOR row.size
        base [index] := (-(in.buf[index][pivot.col] /
                                             pivot.row.value[pivot.col]))
```

```
    IF
      (pivot.row >= pre.size) AND (pivot.row < full.size)
        base [pivot.row MINUS pre.size] := (1.0(REAL64) -
                 pivot.row.value[pivot.col]) / pivot.row.value[pivot.col]
      TRUE
        SKIP
    -- compute new matrix using basis column
    SEQ col = 0 FOR col.size
      SEQ row = 0 FOR row.size
        -- COMMENT sparse case

        -- normal case
        in.buf[row][col] := in.buf[row][col] +
                                    (base[row] * pivot.row.value[col])

:

-- PROC computation
PROC computation (CHAN OF ANY from.root, to.root,
                  [][]REAL64 in.buf, INT pre.size, row.size, col.size, status)
  -- vars
  VAL running IS 1:
  REAL64 base.value, epsilon:
  BOOL degenerate:

  SEQ
    full.size := row.size PLUS pre.size
    from.root ? status
    WHILE status = running
      SEQ
        from.root ? pivot.col
        degenerate := TRUE
        WHILE degenerate
          SEQ
            -- find current processor's pivot row
            SEQ
              pivot.row := 0
              SEQ row = 0 FOR row.size
                SEQ
                  IF
                    (in.buf[row][pivot.col] > ZERO)
                      IF
                        in.buf[row][0] < (-ZERO)
                          base[row] := MAX.REAL64
                        TRUE
                          base[row] := in.buf[row][0] / in.buf[row][pivot.col]
                    TRUE
                      base[row] := MAX.REAL64
                  IF
                    base[row] < base[pivot.row]
                      pivot.row := row
                    TRUE
                      SKIP

            -- select pivot row
            base.value  := base[pivot.row]
            [pivot.row.value FROM 0 FOR col.size] :=
                             [in.buf[pivot.row] FROM 0 FOR col.size]

            to.root ! pivot.row PLUS pre.size;
                      [pivot.row.value FROM 0 FOR col.size];
                      base.value

            from.root ? degenerate
            IF
              degenerate
                SEQ
                  from.root ? epsilon
                  SEQ row = 0 FOR row.size
                    SEQ
                      epsilon := epsilon / 2.0(REAL64)
                      in.buf[row][0] := in.buf[row][0] + epsilon
              TRUE
```

```
                SKIP

        -- inverse Matrix
        from.root ? pivot.row; [pivot.row.value FROM 0 FOR col.size]
        inverseMatrix (in.buf, pivot.row.value,
                    pivot.row, pivot.col, pre.size, row.size, col.size)

        from.root ? status
:

SEQ
  WHILE TRUE
    SEQ
      -- initial data receive
      SEQ row = 0 FOR max.module
        SEQ col = 0 FOR max.col
          in.buf[row][col] := 0.0 (REAL64)

      from.root ? pre.size; row.size; modular
      from.root ? kchar
      WHILE (kchar <> ft.terminated)
        SEQ
          from.root ? row; col; in.data
          in.buf[row MINUS pre.size][col] := in.data
          from.root ? kchar

      -- phase I
      from.root ?  col.size
      computation (from.root, to.root,
                                 in.buf, pre.size, row.size, col.size, status)

      IF
        status = 0
          SEQ
            -- when the artificial variable is in the basis
            from.root ? kchar
            WHILE (kchar <> ft.terminated)
              SEQ
                from.root ? pivot.row
                IF
                  (pivot.row >= pre.size) AND (pivot.row<(pre.size PLUS row.size))
                    [pivot.row.value FROM 0 FOR col.size] :=
                            [in.buf[pivot.row MINUS pre.size] FROM 0 FOR col.size]
                  TRUE
                    SKIP
                to.root ! [pivot.row.value FROM 0 FOR col.size]
                from.root ? redundant
                IF
                  redundant
                    SKIP
                  TRUE
                    SEQ
                      from.root ? [pivot.row.value FROM 0 FOR col.size]; pivot.col
                      inverseMatrix (in.buf, pivot.row.value,
                               pivot.row, pivot.col, pre.size, row.size, col.size)
                from.root ? kchar

            -- phase II
            SEQ
              from.root ? col.size
              -- receive cost vector to sub processors for new object function
              from.root ? [basic FROM 0 FOR col.size]; size
              SEQ index = 0 FOR row.size
                from.root ? cost.buf[index]

              -- form a new cost vector
              REAL64 sum:
              SEQ index1 = 0 FOR col.size
                IF
                  basic[index1] = FALSE
                    SEQ
                      sum := 0.0 (REAL64)
                      SEQ index2 = 0 FOR row.size
```

```
                          sum := sum +
                                (in.buf[index2][index1] * cost.buf[index2])
                        to.root ! sum
                    TRUE
                      SKIP

                computation (from.root, to.root,
                                      in.buf, pre.size, row.size, col.size, status)

              -- send solution
              SEQ
                kchar := 0
                SEQ row = 0 FOR row.size
                  to.root ! kchar; (row PLUS pre.size); in.buf[row][0]
                to.root ! ft.terminated

          TRUE
            SKIP
:
```

A.3 Transputer Network Configuration

```
-- vars
-- define hard link value
VAL link0out IS 0 :
VAL link1out IS 1 :
VAL link2out IS 2 :
VAL link3out IS 3 :
VAL link0in  IS 4 :
VAL link1in  IS 5 :
VAL link2in  IS 6 :
VAL link3in  IS 7 :

-- define soft link (sequential)
VAL root.link.in  IS [link0in,  link2in,  link2in,  link2in,
                      link0in,  link2in,  link2in,  link2in,
                      link0in,  link2in,  link2in,  link2in,
                      link0in,  link2in,  link2in,  link2in]:
VAL root.link.out IS [link0out, link2out, link2out, link2out,
                      link0out, link2out, link2out, link2out,
                      link0out, link2out, link2out, link2out,
                      link0out, link2out, link2out, link2out]:
VAL sub.link.in   IS [link3in,  link3in,  link3in,  link1in,
                      link3in,  link3in,  link3in,  link1in,
                      link3in,  link3in,  link3in,  link1in,
                      link3in,  link3in,  link3in,  link1in]:
VAL sub.link.out  IS [link3out, link3out, link3out, link1out,
                      link3out, link3out, link3out, link1out,
                      link3out, link3out, link3out, link1out,
                      link3out, link3out, link3out, link1out]:

-- COMMENT define soft link (tree)


VAL max.cpu    IS 16:
[max.cpu]CHAN OF ANY from.root, to.root:

PLACED PAR
  PLACED PAR cpu = 0 FOR (max.cpu - 1)
    PROCESSOR cpu T4
      PLACE from.root[cpu]    AT root.link.in[cpu] :
      PLACE to.root   [cpu]    AT root.link.out[cpu]:
      PLACE to.root   [cpu+1] AT sub.link.in[cpu] :
      PLACE from.root[cpu+1] AT sub.link.out[cpu]:
      node (cpu, from.root[cpu], to.root[cpu], to.root[cpu+1], from.root[cpu+1])
  PROCESSOR (max.cpu - 1) T4
    PLACE from.root[max.cpu-1]    AT root.link.in [max.cpu-1] :
    PLACE to.root   [max.cpu-1]    AT root.link.out[max.cpu-1]:
    lastnode (max.cpu-1, from.root[max.cpu-1], to.root[max.cpu-1])
```