

PARALLELIZATION OF
AN INTERIOR POINT ALGORITHM
FOR LINEAR PROGRAMMING

A THESIS SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING AND
INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

by

Müseyin Simitçi

July 1994

THESIS

T

57.74

.556

1994

T
57.74
S56
1994

B 024049

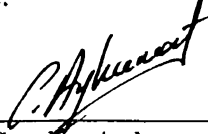
PARALLELIZATION OF
AN INTERIOR POINT ALGORITHM
FOR LINEAR PROGRAMMING

A THESIS SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING AND
INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

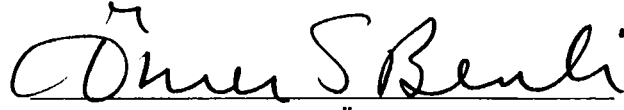
by
Hüseyin Simitçi
July 1994

Hüseyin Simitçi
tarafından hazırlanmıştır.

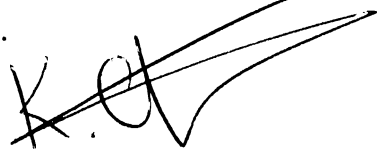
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Asst. Prof. Cevdet Aykanat (Principal Advisor)

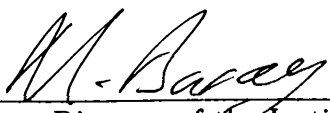
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Assoc. Prof. Ömer Benli

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Asst. Prof. Kemal Ofazer

Approved by the Institute of Engineering:


Prof. Mehmet Baray, Director of the Institute of Engineering

ABSTRACT

PARALLELIZATION OF AN INTERIOR POINT ALGORITHM FOR LINEAR PROGRAMMING

Hüseyin Simitçi

M.S. in Computer Engineering and Information Science

Supervisor: Asst. Prof. Cevdet Aykanat

July 1994

In this study, we present the parallelization of Mehrotra's predictor-corrector interior point algorithm, which is a Karmarkar-type optimization method for linear programming. Computation types needed by the algorithm are identified and parallel algorithms for each type are presented. The repeated solution of large symmetric sets of linear equations, which constitutes the major computational effort in Karmarkar-type algorithms, is studied in detail. Several forward and backward solution algorithms are tested, and buffered backward solution algorithm is developed. Heuristic bin-packing algorithms are used to schedule sparse matrix-vector product and factorization operations. Algorithms having the best performance results are used to implement a system to solve linear programs in parallel on multicomputers. Design considerations and implementation details of the system are discussed, and performance results are presented from a number of real problems.

Keywords: Linear Programming, Interior Point Algorithms, Distributed Systems, Parallel Processing.

ÖZET

BİR İÇ NOKTA DOĞRUSAL PROGRAMLAMA ALGORİTMASININ PARALELLEŞTİRİLMESİ

Hüseyin Simitçi

Bilgisayar ve Enformatik Mühendisliği Bölümü, Yüksek Lisans

Tez Yöneticisi: Asst. Prof. Dr. Cevdet Aykanat

Temmuz 1994

Bu çalışmada, Karmarkar-tipi bir doğrusal programlama optimizasyon algoritması olan Mehrotra'nın predictor-corrector iç nokta algoritmasının paralelleştirilmesi sunulmaktadır. Algoritmanın içerdiği işlem tipleri belirlenmiş ve her işlem tipi için paralel algoritmalar sunulmuştur. Karmarkar-tipi algoritmaların işlem ağırlığını oluşturan büyük simetrik doğrusal denklem kümelerinin çözümü detaylı incelenmiştir. Birçok ileri ve geri çözüm algoritması test edilmiş, bir biriktirmeli geri çözüm algoritması geliştirilmiştir. Seyrek matrisvektör çarpımı ve faktörizasyon işlemlerinin dağıtımı için sezgisel bin-packing algoritmaları kullanılmıştır. Performans sonuçları en iyi olan algoritmalar doğrusal programlama problemlerinin çoklu bilgisayarlarda paralel çözümü için bir sistem geliştirilmesinde kullanılmıştır. Dizayn kısıtları ve uygulama detayları tartışılmış, bazı performans sonuçları sunulmuştur.

Anahtar Kelimeler: Doğrusal Programlama, İç Nokta Algoritmaları, Dağıtık Sistemler, Paralel İşleme.

ACKNOWLEDGEMENTS

I wish to express deep gratitude to my supervisor Asst. Prof. Cevdet Aykanat for his guidance and support during the development of this thesis.

I would also like to thank Assoc. Prof. Ömer Benli and Asst. Prof. Kemal Oflazer for reading and commenting on the thesis.

It is a great pleasure to express my thanks to my family for providing morale support and to all my friends for their invaluable discussions and helps.

Contents

1	Introduction	1
2	Predictor-Corrector Method	5
2.1	Linear Programming Problem	5
2.2	Primal-Dual Path Following Algorithm	6
2.3	Mehrotra's Predictor-Corrector Method	8
2.4	Predictor-Corrector Interior Point Algorithm	9
2.5	Computation Types for PCIPA	10
3	Parallelization of PCIPA	14
3.1	Sparse Matrix-Vector Products	16
3.2	Vector Operations	18
3.3	Sparse Matrix-Matrix Product	19
3.4	Scalar Operations	20
3.5	Sparse Linear System Solution	21
4	Parallel Sparse Cholesky Factorization	23
4.1	Sequential Algorithms	23
4.1.1	Ordering	25
4.1.2	Symbolic Factorization	26
4.1.3	Numeric Factorization	27
4.1.4	Triangular Solution	30
4.2	Parallel Algorithms	30
4.2.1	Ordering and Symbolic Factorization	31
4.2.2	Task Partitioning and Scheduling	32
4.2.3	Numeric Factorization	36
4.2.4	Triangular Solution	39
5	Computational Results for PLOP	51

List of Tables

5.1	Statistics for the NETLIB problems used in this work.	54
5.2	Change of factorization times and separator sizes by balance ratio. (For 16 processors)	54
5.3	Change of (FS+BS) times by balance ratio. (For 16 processors)	54
5.4	Computation times and speed-up values for fan-in factorization.	55
5.5	Computation times and speed-up values for fan-in FS (FIFS). .	56
5.6	Computation times and speed-up values for EBFS.	57
5.7	Computation times and speed-up values for EBBS.	58
5.8	Computation times and speed-up values for send-forward BS (SFBS).	59
5.9	Computation times and speed-up values for buffered BS (BFBS).	60
5.10	Computation times and speed-up values for the computation of matrix-vector product $\mathbf{A}\delta$	61
5.11	Computation times (T,in milliseconds), speed-up (S) and percent efficiency (E) values for the computation of matrix-matrix product $\mathbf{A}\mathbf{D}\mathbf{A}^T$	62
5.12	Computation times (T,in milliseconds), speed-up (S) and percent efficiency (E) values for one iteration of PCIPA (PLOP). .	63

List of Figures

2.1	The primal-dual path following algorithm.	7
2.2	Variables of PCIPA.	10
2.3	Predictor-corrector interior point algorithm, first part.	12
2.4	Predictor-corrector interior point algorithm, second part.	13
3.1	Algorithm to map rows of \mathbf{A} matrix to processors for the multiplication $\mathbf{A}\delta$	17
3.2	Node algorithm for parallel sparse matrix-vector product $\mathbf{A}\delta$	18
3.3	Node algorithm for parallel sparse matrix-vector product $\mathbf{A}^T\psi$	18
3.4	Algorithm to construct product sets for each k_{ij}	20
3.5	Node algorithm for the matrix-matrix product $\mathbf{A}\mathbf{D}\mathbf{A}^T$	20
4.1	Nonzero Entries of $\mathbf{A}\mathbf{D}\mathbf{A}^T$ matrix for <code>woodw</code> problem.	25
4.2	Nonzero Entries of ordered \mathbf{L} (factor) matrix for <code>woodw</code> problem.	26
4.3	Sparse column-Cholesky factorization.	28
4.4	Sparse submatrix-Cholesky factorization.	28
4.5	Elimination tree for <code>woodw</code>	29
4.6	Algorithm for predicting the number of floating-point operations required to generate each column of \mathbf{L}	33
4.7	Algorithm to generate tree weights.	34
4.8	Task Scheduling Algorithm.	35
4.9	Algorithm to partition the tree.	35
4.10	Fan-in Cholesky factorization algorithm.	37
4.11	Spacetime diagram and utilization count for fan-in factorization of <code>80bau3b</code> problem.	38
4.12	Node algorithm for EBFS with $\mathcal{P} = n$	40
4.13	Spacetime diagram and utilization count for EBFS on <code>80bau3b</code> problem.	41
4.14	Fan-in FS algorithm (FIFS).	42

4.15	Spacetime diagram and utilization count for fan-in FS on 80bau3b problem.	43
4.16	Node algorithm for elimination tree based BS (EBBS).	44
4.17	Spacetime diagram and utilization count for EBBS on 80bau3b problem.	45
4.18	Broadcast BS algorithm.	46
4.19	Send-forward BS (SFBS) algorithm.	46
4.20	Buffered BS algorithm (BFBS).	49
4.21	Spacetime diagram and utilization count for BFBS on 80bau3b problem.	50
5.1	Speed-up curves for fan-in factorization algorithm.	64
5.2	Speed-up curves for FIFS algorithm.	64
5.3	Speed-up curves for EBFS algorithm.	65
5.4	Speed-up curves for EBBS algorithm.	65
5.5	Speed-up curves for SFBS algorithm.	66
5.6	Speed-up curves for BFBS algorithm.	66
5.7	Speed-up curves for matrix-vector product $\mathbf{A}\delta$	67
5.8	Speed-up curves for matrix-matrix product $\mathbf{A}\mathbf{D}\mathbf{A}^T$	67
5.9	Speed-up curves for one iteration of PLOP.	68

Chapter 1

Introduction

A consensus is forming that high-end computing will soon be dominated by parallel architectures with a large number of processors which offer huge amounts of computing power. It is inviting to consider harnessing this power to solve difficult optimization problems arising in operations research (OR). Such problems could involve extending existing models to larger time or space domains, adding stochastic elements to previously deterministic models, integrating previously separate or weakly-linked models, or simply tackling difficult combinatorial problems [7]. Another attractive possibility is that of solving in minutes or seconds problems which now take hours or days. Such a capability would allow presently unwieldy models to be run in real time to respond to rapidly changing conditions, and to be used much more freely in “what if” analyses.

The main difficulty is the present lack of sufficiently general parallel sparse linear algebra primitives to permit the wholesale adaptation of existing, proven, continuous optimization methods to parallel architectures.

This work describes the application of highly parallel computing to numerical optimization problems, in particular linear programming which is central to the practice of operations research.

Before getting into details, we should address the question: What is a linear program ? It is an optimization problem in which one wants to minimize or maximize a linear function (of a large number of variables) subject to a large number of linear equality and inequality constraints. For example, linear programming may be used in manufacturing-production planning

to develop a production schedule that meets demand and workstation capacity constraints, while minimizing production costs. It is being used by the U.S Air Force Military Airlift Command (MAC) to solve critical logistics problems and by commercial air-lines to solve scheduling problems, such as crew planning [4].

Research in linear programming methods has been very active since the discovery of the simplex method by Dantzig in the late 1940's. It has long been known that the number of iterations required by the simplex method can grow exponentially in the problem dimension in the worst case. More recently, interest in linear programming methods has been rekindled by the discovery of the projective method of Karmarkar, which has been proven to require $O(nL)$ iterations and $O(n^{3.5})$ time overall, where L is a measure of the size of the problem, and the claims that variants of this method are much faster than the simplex method in practice [1, 6, 19].

In the past few years, there have been many other interior methods proposed. In this work we will concentrate on the parallel implementation of the one proposed by Mehrotra [18], namely predictor-corrector interior point method.

In general, there is a technical barrier to easy application of parallel computing technology to large-scale OR/optimization models — the lack of sufficiently general, highly parallel sparse linear algebra primitives. If one examines the “standard” algorithms of numerical optimization several critical operations appear repeatedly. Perhaps the most common is the factoring of large sparse matrices, this operation forms the bulk of the work in most Newton-related methods, including interior point methods for linear programming. It is clear that without an efficient parallel version of sparse factoring, it will not be practical to simply “port” existing general-purpose sparse optimization codes to parallel architectures.

Previously, most of the research on parallel solution of linear programs concentrated on decomposable linear programs [15, 28], which are assumed to have a block-angular structure. Panwar and Mazumder [25] present a parallel Karmarkar algorithm for orthogonal tree networks. Li *et al.* [21] give parallel algorithms for simplex and revised simplex algorithms. Currently we are not

aware of any research on the parallelization of interior point algorithms for general linear programming problems on distributed memory message-passing parallel computers.

Distributed memory, message-passing parallel computers, which are usually named as *multicomputers*, are the most promising general purpose high performance computers. Implementation of parallel algorithms on multicomputers requires distribution of data and operations among processors while supplying communication and synchronization through messages. For these architectures it is important to explore these capabilities fully to achieve maximum performance. In the following chapters we investigate the efficient parallelization of the most time-consuming steps of interior point (Karmarkar-type) methods and elaborate on the methodology used to develop parallel algorithms for these problems. The algorithms and the methodology are used to implement PLOP (Parallel Linear Optimizer), a parallel implementation of Mehrotra's predictor-corrector interior point algorithm (PCIPA), on iPSC/2 hypercube multicomputer.

In Chapter 2 we will introduce linear programming and PCIPA, and discuss the computation types needed by this algorithm. Parallelization of PCIPA will be discussed in Chapter 3. In this chapter every section will introduce the parallel algorithms for different computation types of PCIPA and how they are implemented in PLOP.

As noted earlier, the main computational effort of the Karmarkar-type linear programming methods involve the repeated solution of large symmetric sets of linear equations. For this reason, algorithms that optimize the PLOP performance for the solution of sparse sets of linear equations are presented separately in Chapter 4. These algorithms involve both Cholesky factorization and forward-and-backward substitution for the solution of linear equations and exploit data locality and concurrency by scheduling the operations among the processors.

Experimental results for PLOP performance obtained from actual runs on linear programs from the NETLIB suite [8] are presented in Chapter 5. These

data represent realistic problems in industry applications ranging from small-scale to large-scale. Finally we derive the conclusions from the earlier discussions in Chapter 6.

Chapter 2

Predictor-Corrector Method

2.1 Linear Programming Problem

Linear programming problem is the problem of minimizing (or maximizing) a linear function subject to a finite number of linear constraints and to the condition that all variables must be nonnegative. The function to be optimized in the linear programming problems is called the *objective function*.

Mathematically, the linear programming problems can be formulated as follows¹:

$$\begin{aligned} & \text{minimize} && \mathbf{c}'\mathbf{x} \\ & \text{subject to} && \mathbf{Ax} = \mathbf{b}, \\ & && \mathbf{x} \geq 0. \end{aligned} \tag{2.1}$$

where \mathbf{A} is a given $m \times n$ matrix and $m < n$, \mathbf{b} is a given m -component right hand side vector, \mathbf{c} is a given n -component coefficient vector that defines the objective function, and \mathbf{x} is an n -component nonnegative unknown vector. Here, m is the number of constraints and n is the number of variables. This linear programming problem is called the *primal problem*.

Associated with this primal problem is the *dual* linear program:

$$\begin{aligned} & \text{maximize} && \mathbf{b}'\mathbf{y} \\ & \text{subject to} && \mathbf{A}^T\mathbf{y} + \mathbf{z} = \mathbf{c}, \\ & && \mathbf{z} \geq 0. \end{aligned} \tag{2.2}$$

¹In this work matrices and vectors are denoted by capital bold letters and small bold letters, respectively. Nonbold small letters are used for scalar values.

which we have written in equality form by introducing slack variables \mathbf{z} (also called *reduced costs*).

The linear programming problem is to find the optimal value of the objective function subject to the linear constraints. The optimization problem arises because the linear equations $\mathbf{Ax} = \mathbf{b}$ is underconstrained, i.e., the coefficient matrix \mathbf{A} contains many more columns (variables) than rows (constraints).

Since Karmarkar's fundamental paper [19] appeared in 1984, many interior point methods for linear programming have been proposed. Among these variants the primal affine-scaling, dual affine-scaling [2, 24], one-phase primal-dual affine-scaling, and one-phase primal-dual path following methods are the most popular ones.

We focus on one variant — Mehrotra's predictor-corrector interior point algorithm (PCIPA). But before introducing PCIPA we will describe the one-phase primal-dual path following algorithm (PDPF), which constitutes the theoretical base of PCIPA. In the next section we give a brief review of the PDPF algorithm. For a more detailed description, see [18, 23, 26]. Also [22] gives an informal and intuitive description.

2.2 Primal-Dual Path Following Algorithm

The PDPF algorithm (Fig. 2.1) is motivated by applying the logarithmic barrier function to eliminate the inequality constraints in (2.1) and (2.2).

Fix $\mu > 0$. Let $(\mathbf{x}_\mu, \mathbf{y}_\mu, \mathbf{z}_\mu)$ be a solution to the following system of $2n + m$ (nonlinear) equations in $2n + m$ unknowns:

$$\mathbf{Ax} = \mathbf{b}, \tag{2.3}$$

$$\mathbf{A}^T \mathbf{y} + \mathbf{z} = \mathbf{c}, \tag{2.4}$$

$$\mathbf{XZ}\mathbf{e} = \mu\mathbf{e}, \tag{2.5}$$

where \mathbf{e} denotes the n -vector of ones, and \mathbf{X} and \mathbf{Z} denote the diagonal matrices with \mathbf{x} and \mathbf{z} vectors, on the diagonal, respectively. The first m equations in (2.3) are part of the primal feasibility requirement and the next n equations in (2.4) are part of the dual feasibility requirement. The last n equations are called μ -complementarity.

```

start with  $\mu > 0, \mathbf{x} > 0, \mathbf{z} > 0$ .
while stopping criterion not satisfied do
     $\boldsymbol{\rho} = \mathbf{b} - \mathbf{A}\mathbf{x}$ ;
     $\boldsymbol{\sigma} = \mathbf{c} - \mathbf{A}^T\mathbf{y} - \mathbf{z}$ ;
     $\boldsymbol{\gamma} = \mu\mathbf{X}^{-1}\mathbf{e} - \mathbf{Z}\mathbf{e}$ ;
     $\mathbf{D} = \mathbf{X}\mathbf{Z}^{-1}$ ;
     $\Delta\mathbf{y} = (\mathbf{A}\mathbf{D}\mathbf{A}^T)^{-1}(\mathbf{A}\mathbf{D}(\boldsymbol{\sigma} - \boldsymbol{\gamma}) + \boldsymbol{\rho})$ ;
     $\Delta\mathbf{z} = \boldsymbol{\sigma} - \mathbf{A}^T\Delta\mathbf{y}$ ;
     $\Delta\mathbf{x} = \mathbf{D}(\boldsymbol{\gamma} - \Delta\mathbf{z})$ ;
    find  $\alpha_p$  and  $\alpha_d$  such that
         $\mathbf{x} > 0$  and  $\mathbf{z} > 0$  is preserved;
     $\mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}/\alpha_p$ ;
     $\mathbf{y} \leftarrow \mathbf{y} + \Delta\mathbf{y}/\alpha_d$ ;
     $\mathbf{z} \leftarrow \mathbf{z} + \Delta\mathbf{z}/\alpha_d$ 

```

Figure 2.1: The primal-dual path following algorithm.

Applying Newton's method to (2.3)-(2.5), we get

$$\mathbf{A}\Delta\mathbf{x} = \boldsymbol{\rho}, \quad (2.6)$$

$$\mathbf{A}^T\Delta\mathbf{y} + \Delta\mathbf{z} = \boldsymbol{\sigma}, \quad (2.7)$$

$$\mathbf{Z}\Delta\mathbf{x} + \mathbf{X}\Delta\mathbf{z} = \boldsymbol{\phi}, \quad (2.8)$$

where

$$\boldsymbol{\rho} = \mathbf{b} - \mathbf{A}\mathbf{x},$$

$$\boldsymbol{\sigma} = \mathbf{c} - \mathbf{A}^T\mathbf{y} - \mathbf{z},$$

and

$$\boldsymbol{\phi} = \mu\mathbf{e} - \mathbf{X}\mathbf{Z}\mathbf{e}.$$

If we multiply (2.7) by $\mathbf{A}\mathbf{X}\mathbf{Z}^{-1}$ and then use (2.8) followed by (2.6), we see that

$$\Delta\mathbf{y} = (\mathbf{A}\mathbf{D}\mathbf{A}^T)^{-1}(\mathbf{A}\mathbf{D}(\boldsymbol{\sigma} - \boldsymbol{\gamma}) + \boldsymbol{\rho}), \quad (2.9)$$

where \mathbf{D} is the positive definite diagonal matrix satisfying

$$\mathbf{D} = \mathbf{X}\mathbf{Z}^{-1}$$

and

$$\boldsymbol{\gamma} = \mathbf{X}^{-1}\boldsymbol{\phi} = \mu\mathbf{X}^{-1}\mathbf{e} - \mathbf{Z}\mathbf{e}.$$

Once $\Delta\mathbf{y}$ is known, it is easy to solve for $\Delta\mathbf{z}$ and $\Delta\mathbf{x}$,

$$\Delta\mathbf{z} = \boldsymbol{\sigma} - \mathbf{A}^T\Delta\mathbf{y} \quad (2.10)$$

and

$$\Delta \mathbf{x} = \mathbf{D}(\boldsymbol{\gamma} - \Delta \mathbf{z}). \quad (2.11)$$

The desired update is then

$$\mathbf{x} \leftarrow \mathbf{x} + \Delta \mathbf{x}, \quad (2.12)$$

$$\mathbf{y} \leftarrow \mathbf{y} + \Delta \mathbf{y}, \quad (2.13)$$

$$\mathbf{z} \leftarrow \mathbf{z} + \Delta \mathbf{z}. \quad (2.14)$$

However, there is no guarantee that this update will preserve the nonnegativity of \mathbf{x} and \mathbf{z} , so a shorter step has to be taken to keep $\mathbf{x} > 0$ and $\mathbf{z} > 0$ using ratio tests.

2.3 Mehrotra's Predictor-Corrector Method

Mehrotra introduces a power series variant of the primal-dual algorithm. The method again uses the logarithmic barrier method to derive the first-order conditions (2.3)-(2.5). Rather than applying Newton's method to (2.3)-(2.5) to generate correction terms to the current estimate, we substitute the new point into (2.3)-(2.5) directly, yielding

$$\mathbf{A}(\mathbf{x} + \Delta \mathbf{x}) = \mathbf{b}, \quad (2.15)$$

$$\mathbf{A}^T(\mathbf{y} + \Delta \mathbf{y}) + (\mathbf{z} + \Delta \mathbf{z}) = \mathbf{c}, \quad (2.16)$$

$$(\mathbf{X} + \Delta \mathbf{X})(\mathbf{Z} + \Delta \mathbf{Z})\mathbf{e} = \mu \mathbf{e}, \quad (2.17)$$

where $\Delta \mathbf{X}$ and $\Delta \mathbf{Z}$ are diagonal matrices having elements $\Delta \mathbf{x}$ and $\Delta \mathbf{z}$, respectively. Simple algebra reduces (2.15)-(2.17) to the equivalent system

$$\mathbf{A}\Delta \mathbf{x} = \mathbf{b} - \mathbf{A}\mathbf{x}, \quad (2.18)$$

$$\mathbf{A}^T \Delta \mathbf{y} + \Delta \mathbf{z} = \mathbf{c} - \mathbf{A}^T \mathbf{y} - \mathbf{z}, \quad (2.19)$$

$$\mathbf{X}\Delta \mathbf{z} + \mathbf{Z}\Delta \mathbf{x} = \mu \mathbf{e} - \mathbf{X}\mathbf{Z}\mathbf{e} - \Delta \mathbf{X}\Delta \mathbf{Z}\mathbf{e}. \quad (2.20)$$

The left-hand side of (2.18)-(2.20) is identical to (2.6)-(2.8), while the right-hand side has a distinct difference. There is a non-linear term $\Delta \mathbf{X}\Delta \mathbf{Z}\mathbf{e}$ in the right-hand side of (2.20). To determine a step approximately satisfying (2.18)-(2.20), Mehrotra suggests first solving the defining equations for the primal-dual affine direction:

$$\mathbf{A}\Delta \hat{\mathbf{x}} = \mathbf{b} - \mathbf{A}\mathbf{x}, \quad (2.21)$$

$$\mathbf{A}^T \Delta \hat{\mathbf{y}} + \Delta \hat{\mathbf{z}} = \mathbf{c} - \mathbf{A}^T \mathbf{y} - \mathbf{z}, \quad (2.22)$$

$$\mathbf{X}\Delta \hat{\mathbf{z}} + \mathbf{Z}\Delta \hat{\mathbf{x}} = -\mathbf{X}\mathbf{Z}\mathbf{e}. \quad (2.23)$$

These directions are then used in two distinct ways: to approximate the non-linear terms in the right-hand side of (2.18)-(2.20) and to dynamically estimate μ . The actual new step $\Delta \mathbf{x}, \Delta \mathbf{y}, \Delta \mathbf{z}$ is then chosen as the solution to

$$\mathbf{A}\Delta \mathbf{x} = \mathbf{b} - \mathbf{A}\mathbf{x}, \quad (2.24)$$

$$\mathbf{A}^T \Delta \mathbf{y} + \Delta \mathbf{z} = \mathbf{c} - \mathbf{A}^T \mathbf{y} - \mathbf{z}, \quad (2.25)$$

$$\mathbf{X}\Delta \mathbf{z} + \mathbf{Z}\Delta \mathbf{x} = \mu \mathbf{e} - \mathbf{X}\mathbf{Z}\mathbf{e} - \Delta \hat{\mathbf{X}}\Delta \hat{\mathbf{Z}}\mathbf{e}. \quad (2.26)$$

Clearly, all that has changed from (2.6)-(2.8) is the right-hand side, so the matrix algebra remains the same as in the solutions (2.9),(2.10) and (2.11). Ratio tests are now done using $\Delta \mathbf{x}, \Delta \mathbf{y}$ and $\Delta \mathbf{z}$ to determine actual step sizes α_p and α_d , and the actual new point $\mathbf{x}, \mathbf{y}, \mathbf{z}$ is defined by (2.12)-(2.14).

The results given in [18] show that the predictor-corrector method almost always reduces the iteration count and usually reduces computation time. Furthermore, as problem size and complexity increase, the improvements in both iteration count and execution time become greater. Thus the predictor-corrector method is a higher-order method that is generally very computationally efficient.

2.4 Predictor-Corrector Interior Point Algorithm

For the implementation of PCIPA, we will consider a more general form of the primal linear problem with the addition of upper bounds and ranges,

$$\begin{aligned} &\text{minimize} && \mathbf{c}'\mathbf{x} \\ &\text{subject to} && \mathbf{A}\mathbf{x} - \mathbf{w} = \mathbf{b}, \\ &&& \mathbf{x} + \mathbf{t} = \mathbf{u}, \\ &&& \mathbf{w} + \mathbf{p} = \mathbf{r} \\ &&& \mathbf{x}, \mathbf{w}, \mathbf{t}, \mathbf{p} \geq 0. \end{aligned} \quad (2.27)$$

where \mathbf{u} is the upper bound vector, \mathbf{r} is range vector, and $\mathbf{w}, \mathbf{t}, \mathbf{p}$ are appropriate slack variables.

Corresponding to the primal problem in (2.27), dual linear problem is as

m	number of rows (constraints)	n	number of columns (variables)
\mathbf{A}	constraint matrix	\mathbf{b}	right-hand side
\mathbf{c}	objective function	f	fixed adjustment to obj. func.
\mathbf{r}	ranges	ℓ	lower bounds
\mathbf{u}	upper bounds	\mathbf{w}	primal surpluses
\mathbf{x}	primal solution	\mathbf{y}	dual solution
\mathbf{z}	dual slacks	\mathbf{p}	range slacks
\mathbf{q}	dual range slacks	\mathbf{s}	dual for upper bound slacks
\mathbf{t}	upper bound slacks	\mathbf{v}	dual for range (\mathbf{w})
p_{res}	primal residual (i.e., infeasibility)	d_{res}	dual residual (i.e., infeasibility)
p_{obj}	primal objective value	d_{obj}	dual objective value
\mathbf{u}_b	shifted upper bounds		

Figure 2.2: Variables of PCIPA.

follows:

$$\begin{aligned}
& \text{maximize} && \mathbf{b}^t \mathbf{y} - \mathbf{u}^t \mathbf{s} - \mathbf{r}^t \mathbf{q} \\
& \text{subject to} && \mathbf{A}^T \mathbf{y} - \mathbf{s} + \mathbf{z} = \mathbf{c}, \\
& && \mathbf{y} + \mathbf{q} - \mathbf{v} = \mathbf{0}, \\
& && \mathbf{s}, \mathbf{q}, \mathbf{v}, \mathbf{z} \geq \mathbf{0}.
\end{aligned} \tag{2.28}$$

Implementation of Mehrotra's predictor-corrector method on this form of linear problem results in the predictor-corrector interior point algorithm (PCIPA) given in Figs. 2.3 and 2.4. Linear programming variables used in PCIPA are explained in Fig. 2.2. In this algorithm, a variable with a capital letter, other than \mathbf{A} matrix, denotes a diagonal matrix with the corresponding vector on the diagonal. We use “ $.^T$ ” for matrix transpose and “ $.^t$ ” for vector transpose.

Issues concerning the construction of an initial solution and an effective stopping rule are studied in the literature in detail [1, 5, 9, 23].

2.5 Computation Types for PCIPA

A study of PCIPA given in Figs. 2.3 and 2.4 reveals that PCIPA has a wealth of computation types. We will classify them into five general groups:

1. *Sparse Matrix-Vector Products.* There are two kinds of matrix-vector product. One is the product of an $m \times n$ matrix with an n vector ($\mathbf{A}\delta$), the other is the product of an $n \times m$ matrix with an m vector ($\mathbf{A}^T\psi$).
2. *Vector Operations.* These operations include dense vector sum (e.g., $\mathbf{y} + \mathbf{q}$), difference (e.g., $\mathbf{s} - \mathbf{z}$), inner product (e.g., $\mathbf{c}^t \mathbf{x}$). We can also include diagonal matrix operations in this type because they are simply

treated as vectors. For example $\mathbf{Z}\mathbf{X}^{-1}$ is computed as simply element by element division ($\mathbf{z}_i/\mathbf{x}_i$) of two vectors \mathbf{z} and \mathbf{x} .

3. *Sparse Matrix-Matrix Product.* This involves $\mathbf{A}\mathbf{D}\mathbf{A}^T$, which is the product of two sparse matrices \mathbf{A} and \mathbf{A}^T scaled by the elements of the diagonal matrix \mathbf{D} .
4. *Scalar operations.* Besides standard scalar operations, this type involves the search for a minimum or maximum among a list of scalar values.
5. *Sparse Linear System Solution.* This is where the most of the solution time is used in PCIPA and involves the solution of a large sparse linear system $\mathbf{K}\mathbf{x} = \mathbf{b}$, where \mathbf{K} is a positive definite matrix. In PCIPA, \mathbf{K} appears to be $(\mathbf{A}\mathbf{D}_n\mathbf{A}^T + \mathbf{D}_m)$.

In the next chapter, we will discuss the parallelization of these computation types in detail. Since our goal is to get a parallel implementation of the PCIPA, we will have a unified approach in the parallelization of these types by considering the interactions among them.

```

find an initial solution
start with  $\mu > 0$ ;
 $f = \mathbf{c}^t \boldsymbol{\ell}$ ;  $\mathbf{u}_b = \mathbf{u} - \boldsymbol{\ell}$ ;  $\mathbf{b} = \mathbf{b} - \mathbf{A} \boldsymbol{\ell}$ ;
 $n_b = \mathbf{b}^t \mathbf{b}$ ;  $n_c = \mathbf{c}^t \mathbf{c}$ ;
itr = 0;
while itr  $\leq$  iteration-limit do

     $\boldsymbol{\rho} = \mathbf{b} - \mathbf{A} \mathbf{x} + \mathbf{w}$ ;  $\boldsymbol{\sigma} = \mathbf{c} - \mathbf{A}^T \mathbf{y} + \mathbf{s} - \mathbf{z}$ ;

     $\boldsymbol{\alpha} = \mathbf{r} - \mathbf{w} - \mathbf{p}$ ;  $\boldsymbol{\beta} = \mathbf{y} + \mathbf{q} - \mathbf{v}$ ;

     $\boldsymbol{\tau} = \mathbf{u}_b - \mathbf{x} - \mathbf{t}$ ;

     $p_{obj} = \mathbf{c}^t \mathbf{x} + f$ ;  $p_{res} = \sqrt{\boldsymbol{\rho}^t \boldsymbol{\rho} + \boldsymbol{\tau}^t \boldsymbol{\tau} + \boldsymbol{\alpha}^t \boldsymbol{\alpha}} / (n_b + 1)$ ;

     $d_{obj} = \mathbf{b}^t \mathbf{y} - \mathbf{u}_b^t \mathbf{s} - \mathbf{r}^t \mathbf{q} + f$ ;  $d_{res} = \sqrt{\boldsymbol{\sigma}^t \boldsymbol{\sigma} + \boldsymbol{\beta}^t \boldsymbol{\beta}} / (n_c + 1)$ ;

    if stopping-rule()
        optimal found; break;

     $\mathbf{D}_n = (\mathbf{Z} \mathbf{X}^{-1} + \mathbf{S} \mathbf{T}^{-1})^{-1}$ ;  $\mathbf{D}_m = (\mathbf{Y} \mathbf{W}^{-1} + \mathbf{Q} \mathbf{P}^{-1})^{-1}$ ;

     $\boldsymbol{\gamma}_z = -\mathbf{z}$ ;  $\boldsymbol{\gamma}_w = -\mathbf{w}$ ;  $\boldsymbol{\gamma}_t = -\mathbf{t}$ ;  $\boldsymbol{\gamma}_q = -\mathbf{q}$ ;

     $\boldsymbol{\sigma} = \boldsymbol{\sigma} - \boldsymbol{\gamma}_z$ ;  $\boldsymbol{\rho} = \boldsymbol{\rho} + \boldsymbol{\gamma}_w$ ;  $\boldsymbol{\tau} = \boldsymbol{\tau} - \boldsymbol{\gamma}_t$ ;  $\boldsymbol{\alpha} = \boldsymbol{\alpha} - \boldsymbol{\gamma}_w$ ;  $\boldsymbol{\beta} = \boldsymbol{\beta} - \boldsymbol{\gamma}_q$ ;

     $\Delta \mathbf{y} = (\mathbf{A} \mathbf{D}_n \mathbf{A}^T + \mathbf{D}_m)^{-1} (\boldsymbol{\rho} + \mathbf{D}_m (\mathbf{Q} \mathbf{P}^{-1} \boldsymbol{\alpha} - \boldsymbol{\beta}) + \mathbf{A} \mathbf{D}_n (\boldsymbol{\sigma} - \mathbf{S} \mathbf{T}^{-1} \boldsymbol{\tau}))$ ;

     $\Delta \mathbf{x} = \mathbf{D}_n (\mathbf{A}^T \Delta \mathbf{y} - \boldsymbol{\sigma} + \mathbf{S} \mathbf{T}^{-1} \boldsymbol{\tau})$ ;

     $\Delta \mathbf{s} = \mathbf{S} (\Delta \mathbf{x} - \boldsymbol{\tau}) \mathbf{T}^{-1} \mathbf{e}$ ;  $\Delta \mathbf{p} = \mathbf{D}_m (\Delta \mathbf{y} + \mathbf{V} \mathbf{W}^{-1} \boldsymbol{\alpha} + \boldsymbol{\beta})$ ;

     $\Delta \mathbf{v} = \mathbf{D}_m \mathbf{V} \mathbf{W}^{-1} (\Delta \mathbf{y} - \mathbf{Q} \mathbf{P}^{-1} \boldsymbol{\alpha} + \boldsymbol{\beta})$ ;

     $\Delta \mathbf{z} = \boldsymbol{\gamma}_z - \mathbf{Z} \mathbf{X}^{-1} \Delta \mathbf{x}$ ;  $\Delta \mathbf{w} = \boldsymbol{\gamma}_w - \mathbf{W} \mathbf{V}^{-1} \Delta \mathbf{v}$ ;

     $\Delta \mathbf{t} = \boldsymbol{\gamma}_t - \mathbf{T} \mathbf{S}^{-1} \Delta \mathbf{s}$ ;  $\Delta \mathbf{q} = \boldsymbol{\gamma}_q - \mathbf{Q} \mathbf{P}^{-1} \Delta \mathbf{p}$ ;

     $\alpha_p = \max_i \{1, \frac{-1}{0.95} \min\{\Delta \mathbf{x}_i / \mathbf{x}_i, \Delta \mathbf{w}_i / \mathbf{w}_i, \Delta \mathbf{t}_i / \mathbf{t}_i, \Delta \mathbf{p}_i / \mathbf{p}_i\}\}$ ;

     $\alpha_d = \max_i \{1, \frac{-1}{0.95} \min\{\Delta \mathbf{z}_i / \mathbf{z}_i, \Delta \mathbf{y}_i / \mathbf{y}_i, \Delta \mathbf{s}_i / \mathbf{s}_i, \Delta \mathbf{q}_i / \mathbf{q}_i, \Delta \mathbf{v}_i / \mathbf{v}_i\}\}$ ;

     $\alpha_p = \max\{1, \alpha_p, \alpha_d\}$ ;

```

Figure 2.3: Predictor-corrector interior point algorithm, first part.

```


$$\mu = (2(\alpha_p - 1)/(\alpha_p + 20))(s^t t + p^t q + z^t x + v^t w)/(2n + 2m);$$


$$\gamma_z = \mu X^{-1} e - \Delta X X^{-1} \Delta z;$$


$$\gamma_t = \mu S^{-1} e - \Delta S S^{-1} \Delta t;$$


$$\gamma_q = \mu P^{-1} e - \Delta P P^{-1} \Delta q;$$


$$\gamma_w = \mu V^{-1} e - \Delta V V^{-1} \Delta w;$$


$$\sigma = \sigma - \gamma_z; \quad \rho = \rho + \gamma_w; \quad \tau = \tau - \gamma_t; \quad \alpha = \alpha - \gamma_w; \quad \beta = \beta - \gamma_q;$$


$$\gamma_z = \gamma_z - z; \quad \gamma_w = \gamma_w - w; \quad \gamma_t = \gamma_t - t; \quad \gamma_q = \gamma_q - q;$$


$$\Delta y = (A D_n A^T + D_m)^{-1}(\rho + D_m(Q P^{-1} \alpha - \beta) + A D_n(\sigma - S T^{-1} \tau));$$


$$\Delta x = D_n(A^T \Delta y - \sigma + S T^{-1} \tau);$$


$$\Delta s = S(\Delta x - \tau) T^{-1} e; \quad \Delta p = D_m(\Delta y + V W^{-1} \alpha + \beta);$$


$$\Delta v = D_m V W^{-1}(\Delta y - Q P^{-1} \alpha + \beta);$$


$$\Delta z = \gamma_z - Z X^{-1} \Delta x; \quad \Delta w = \gamma_w - W V^{-1} \Delta v;$$


$$\Delta t = \gamma_t - T S^{-1} \Delta s; \quad \Delta q = \gamma_q - Q P^{-1} \Delta p;$$


$$\alpha_p = \max_i \{1, \frac{-1}{0.95} \min\{\Delta x_i/x_i, \Delta w_i/w_i, \Delta t_i/t_i, \Delta p_i/p_i\}\};$$


$$\alpha_d = \max_i \{1, \frac{-1}{0.95} \min\{\Delta z_i/z_i, \Delta y_i/y_i, \Delta s_i/s_i, \Delta q_i/q_i, \Delta v_i/v_i\}\};$$

if  $((c^t \Delta x < 0)$  and  $(\alpha_p = 0)$  and  $(p_{res} \leq 10^{-5}))$ 
    primal unbounded; exit;

if  $((b^t \Delta y < 0)$  and  $(\alpha_d = 0)$  and  $(d_{res} \leq 10^{-5}))$ 
    dual unbounded; exit;

if  $(\alpha_p < 1)$   $\alpha_p = 1$ ; if  $(\alpha_d < 1)$   $\alpha_d = 1$ ;

 $w = w + \Delta w/\alpha_p; \quad x = x + \Delta x/\alpha_p; \quad y = y + \Delta y/\alpha_d;$ 
 $z = z + \Delta z/\alpha_d; \quad s = s + \Delta s/\alpha_d; \quad t = t + \Delta t/\alpha_p;$ 
 $v = v + \Delta v/\alpha_d; \quad p = p + \Delta p/\alpha_p; \quad q = q + \Delta q/\alpha_d;$ 

 $b = b + A \ell;$ 
 $x = x + \ell.$ 

```

Figure 2.4: Predictor-corrector interior point algorithm, second part.

Chapter 3

Parallelization of PCIPA

The purpose of this chapter is to investigate the efficient parallelization of PCIPA on medium-to-coarse grain multicomputers. These architectures have the nice scalability feature due to the lack of shared resources and increasing communication bandwidth with increasing number of processors. In multicomputers, processors have neither shared memory nor shared address space. Synchronization and coordination among processors are achieved through explicit message passing. Each processor can only access its local memory. Processors of a multicomputer are usually connected by utilizing one of the well-known direct interconnection network topologies such as ring, mesh, hypercube and etc.

In order to achieve high efficiency on such architectures, the algorithm must be designed so that both computations and data can be distributed to the processors with local memories in such a way that computational tasks can be run in parallel, balancing the computational loads of the processors. Communication between processors to exchange data must also be considered as part of the algorithm. One important factor in designing parallel algorithms is *granularity*. *Granularity* depends on both the application and the parallel machine. In a parallel machine with high communication latency (start-up time), the algorithm should be structured so that large amounts of computation are done between successive communication steps. That is, both the number and the volume of communications should be minimized in order to reduce the communication overhead. The communication structure of the parallel algorithm is also a crucial issue. In a multicomputer architecture, each adjacent pair of processors can concurrently communicate with each other over the communication

links connecting them. Such communications are referred as *single-hop* communications. However, non-adjacent processors can communicate with each other by means of *software* or *hardware routing*. Such communications are referred as *multi-hop* communications. Multi-hop communications are usually routed in a static manner over the shortest paths of links between the communicating pairs of processors. In software routing, the cost of multi-hop communications is substantially greater than that of the single-hop messages since all intermediate processors on the path are intercepted during the communication. The performance difference between an individual multi-hop and single-hop communication is relatively small in hardware routing. However, a number of concurrent multi-hop communications may congest the routing network thus resulting in substantial performance degradation. Moreover, in almost all commercially available multicomputer architectures, interprocessor communications can only be initiated from/into contiguous local memory locations. Hence, communications from/into scattered memory locations may introduce considerable overhead to the parallel program. In this work, all these points are considered in designing an efficient parallel PCIPA algorithm for multicomputers.

Our parallel linear optimizer (PLOP) program, running PCIPA, consists of two phases. In the preprocessing phase, preparatory work is done such that necessary data structures for the later phase is constructed and distributed to the processors. In the solution phase, PCIPA is executed by all processors in parallel.

All of the matrices used in our implementation are stored in a sparse format. For example, the linear programming coefficient matrix \mathbf{A} is stored column-wise as a sequence of compressed sparse column vectors, which is referred as column-compressed, row-index storage scheme. Within each column, nonzero elements are stored in order of increasing row indices together with their row indices. However, some of the operations in PCIPA require access of rows of \mathbf{A} . For the sake of efficient implementation of these operations matrix \mathbf{A} is also stored row-wise as a duplicate representation. This representation, which is referred as row-compressed, column-index storage scheme, is the dual of the row-wise storage scheme. Operations in PCIPA which require access of rows of \mathbf{A}^T can be efficiently implemented by using the column-wise storage since

column-wise storage of \mathbf{A} corresponds to the row-wise storage of \mathbf{A}^T .

We use $\text{Struct}(\mathbf{M}, k)$ to denote the set of row indices of the nonzero entries¹ in column k of the matrix \mathbf{M} . That is,

$$\text{Struct}(\mathbf{M}, k) = \{i \mid m_{ik} \neq 0\}.$$

Similarly, $\text{Struct}(\mathbf{x})$ denotes the set of indices of the nonzero entries in vector \mathbf{x} . That is,

$$\text{Struct}(\mathbf{x}) = \{i \mid x_i \neq 0\}.$$

We will use $\eta(\mathbf{M})$ to denote the number of nonzero entries in sparse matrix \mathbf{M} and $\eta(\mathbf{M}, i)$ to denote the number of nonzero entries in the i -th column of \mathbf{M} . Here, \mathcal{P} denotes the number of processors used.

Throughout this work, column-wise distribution of sparse matrices is used. Column i of a sparse matrix \mathbf{M} is assigned to the processor $\text{map}(\mathbf{M}, i)$. We use $\text{mycols}(\mathbf{M})$ to denote the set of columns of \mathbf{M} owned by the calling processor. Usually, the $\text{map}(\cdot)$ function will be determined during the preprocessing phase and solution phase is constrained to use these mappings.

3.1 Sparse Matrix-Vector Products

PCIPA involves two types of matrix vector products, $\mathbf{A}\boldsymbol{\delta}$ and $\mathbf{A}^T\boldsymbol{\psi}$ where $\boldsymbol{\delta}$ is an n -vector (e.g., $\boldsymbol{\delta} = \mathbf{x}$ and $\boldsymbol{\delta} = \mathbf{D}_n(\boldsymbol{\sigma} - \mathbf{S}\mathbf{T}^{-1}\boldsymbol{\tau})$) and $\boldsymbol{\psi}$ is an m -vector (e.g., $\boldsymbol{\psi} = \mathbf{y}$ and $\boldsymbol{\psi} = \boldsymbol{\Delta}\mathbf{y}$). To perform these operations in parallel, rows of \mathbf{A} and \mathbf{A}^T matrices should be distributed among processors which corresponds to row-wise and column-wise distribution of \mathbf{A} matrix, respectively. The matrix-vector product $\mathbf{A}\boldsymbol{\delta}$ involves m inner products of sparse row vectors with the n -vector $\boldsymbol{\delta}$. The computational cost of the i -th inner product (which corresponds to row i) is $2\eta(\mathbf{A}^T, i)$ floating point operations, where $\eta(\mathbf{A}^T, i)$ denotes the total number of nonzeros in the i -th row of matrix \mathbf{A} . Hence, in order to achieve the load balance during the parallel execution of this type of operations, row-wise distribution of \mathbf{A} should be such that the sum of the number of nonzero entries of the rows assigned to each processor should be equal as much as possible. Note that, this mapping problem is equivalent to

¹*Nonzero* is used to mean the entries which are structurally not equal to zero, though they can get a zero value during the operations.


```

    binsize = ( $\lfloor m/\mathcal{P} \rfloor$ ) + 1
    for  $i = 1$  to  $m$  do
        roww( $i$ ) =  $\eta(\mathbf{A}^T, i)$ 
    for  $i = 1$  to  $\mathcal{P}$  do
        pweight( $i$ ) = 0
        prownum( $i$ ) = 0
        pstate( $i$ ) = EMPTY
    sort(roww, rowidx)
    {sort row weights in descending order and put
     index values in rowidx. rowidx(newindex) = oldindex.}
    for  $i = m$  downto 1 do
        min = 0
        for  $k = 1$  to  $\mathcal{P}$  do
            if ((pweight( $k$ ) < pweight(min) or pstate(min) = FULL)
                and pstate( $k$ )  $\neq$  FULL )
                min =  $k$ 
        pweight(min) = pweight(min) + roww( $i$ )
        map( $\mathbf{A}$ , rowidx( $i$ )) = min
        prownum(min) = prownum(min) + 1
        if ( prownum(min)  $\geq$  binsize)
            pstate(min) = FULL

```

Figure 3.1: Algorithm to map rows of \mathbf{A} matrix to processors for the multiplication $\mathbf{A}\delta$.

the p -way number partitioning problem which is known to be NP-hard. In this work we employ a bin-packing heuristic for this mapping problem. Every processor is considered as a bin. In every iteration of the mapping algorithm the row with the highest operation count (number of nonzero entries) is assigned to the processor with the lowest weight, until every row is assigned to a processor.

As will be discussed in Section 3.2, row (column) mappings of the \mathbf{A} matrix also determine the distribution of the m -vectors (n -vectors) to that processor. Hence, computational costs of local vector-vector operations are proportional to the number of rows assigned to individual processors. Thus, the number of rows assigned to different processors should be equal as much as possible in order to achieve load balance during the local vector-vector operations. So the objectives of the mapping algorithm can be stated as *the distribution of the rows among the bins such that every bin has almost equal weight and almost equal number of rows*. To achieve these objectives, $\lfloor m/\mathcal{P} \rfloor + 1$ is assigned as the capacities of all bins. During the execution of the algorithm we consider a bin exceeding this maximum bin size as full. Column mapping algorithm is very similar to the row mapping algorithm illustrated in Fig. 3.1.

```

{ $\delta$  and  $\vartheta$  are local vectors.}
perform global-collect operation on local  $\delta$  vectors
to collect the global  $\delta$  vector  $\delta_g$ 
for  $j \in \text{mycols}(\mathbf{A}^T)$  do
     $\vartheta(j) = 0$ 
    for  $k \in \text{Struct}(\mathbf{A}^T, j)$  do
         $\vartheta(j) = \vartheta(j) + a_{kj}^t \times \delta_g(k)$ 

```

Figure 3.2: Node algorithm for parallel sparse matrix-vector product $\mathbf{A}\delta$.

```

{ $\psi$  and  $\nu$  are local vectors.}
perform global-collect operation on local  $\psi$  vectors
to collect the global  $\psi$  vector  $\psi_g$ 
for  $j \in \text{mycols}(\mathbf{A})$  do
     $\nu(j) = 0$ 
    for  $k \in \text{Struct}(\mathbf{A}, j)$  do
         $\nu(j) = \nu(j) + a_{kj} \times \psi_g(k)$ 

```

Figure 3.3: Node algorithm for parallel sparse matrix-vector product $\mathbf{A}^T\psi$.

Algorithm that computes $\mathbf{A}\delta$ is shown in Fig. 3.2. Here, δ is used to denote the local portion of the global δ vector δ_g . In the algorithm every node processor computes inner products of its sparse row vectors with δ_g . In the algorithm $\text{Struct}(\mathbf{A}^T, j)$ denotes the structure of the j -th row of \mathbf{A} . Algorithms for the computation of $\mathbf{A}^T\psi$ (Fig. 3.3) are duals of the ones given for $\mathbf{A}\delta$ and can be derived in the same spirit.

3.2 Vector Operations

All vectors are treated as dense. These dense vectors are distributed among the processors according to the mappings obtained in Section 3.1. Vectors with size n are distributed according to the column mappings and vectors with size m are distributed according to the row mappings obtained for \mathbf{A} . Most of these dense operations doesn't require any communication or synchronization among processors. Every processor computes the corresponding operations with the portions of the vectors it owns. As before we will again use \mathbf{x} to denote the portion of the global \mathbf{x} vector that is owned by a processor.

To compute the vector sum $\mathbf{p} + \mathbf{q}$ in parallel each processor will concurrently compute $\mathbf{p} + \mathbf{q}$ with its local portions. Other vector operations are similarly computed. Computation of the inner product $\mathbf{c}'\mathbf{x}$ requires a global sum operation at the end. After each processor computes its local inner product $\mathbf{c}'\mathbf{x}$, the partial sums should be globally summed.

Diagonal matrices are treated as simply vectors. Hence, operations on diagonal matrices are computed in parallel similar to corresponding vector operations as discussed above.

3.3 Sparse Matrix-Matrix Product

PCIPA and all other types of interior point algorithms requires the formation and factorization of a matrix

$$\mathbf{K} = \mathbf{A}\mathbf{D}\mathbf{A}^T$$

in every iteration, where only diagonal matrix \mathbf{D} changes among the iterations.

In most of the sequential implementations [1, 23] the nonzero elements of the outer products $\mathbf{a}_{*i}\mathbf{a}_{*i}^t$ are stored, where \mathbf{a}_{*i} is the i th column of \mathbf{A} . \mathbf{K} is then calculated as

$$\mathbf{A}\mathbf{D}\mathbf{A}^T = \sum d_i \mathbf{a}_{*i} \mathbf{a}_{*i}^t$$

where the product of d_i is taken with each nonzero element of $\mathbf{a}_{*i}\mathbf{a}_{*i}^t$.

We have to modify this approach to calculate $\mathbf{A}\mathbf{D}\mathbf{A}^T$ in parallel. Since columns of \mathbf{K} will be distributed among the processors, products generated by a column \mathbf{a}_{*i} must be distributed to the processors those need them in every iteration. This would cause a high communication count and volume.

To tackle with this problem, our approach is to locate a nonzero element of \mathbf{K} and the set of products that will be added to it on the same processor. The preprocessing phase is shown in Fig. 3.4. Here $\text{prds}(k_{ij})$ denotes the set of tuples that constitute the products that will be summed to k_{ij} . In a tuple (prd, idx) , prd denotes the product and idx is the index of the d_{idx} element that will be multiplied with it. Because of the reasons that will be explained in the next chapter columns and rows of \mathbf{K} will be reordered, and $\text{perm}(\cdot)$

```

for  $i = 1$  to  $m$  do
  for  $j \in \text{Struct}(\mathbf{K}, i)$  and  $j \geq i$  do
     $\text{prds}(k_{ij}) = \emptyset$ 
for  $i = 1$  to  $n$  do
  for  $\text{row} \in \text{Struct}(\mathbf{A}, i)$  do
    for  $\text{col} \in \text{Struct}(\mathbf{A}, i)$  do
      if  $\text{row} \geq \text{col}$ 
         $\text{col2} = \text{perm}(\text{col})$ 
         $\text{row2} = \text{perm}(\text{row})$ 
         $\text{prd} = a_{\text{col}, i} * a_{\text{row}, i}$ 
        if  $\text{row2} > \text{col2}$ 
           $\text{prds}(k_{\text{row2}, \text{col2}}) = \text{prds}(k_{\text{row2}, \text{col2}}) \cup \{ (\text{prd}, i) \}$ 
        else
           $\text{prds}(k_{\text{col2}, \text{row2}}) = \text{prds}(k_{\text{col2}, \text{row2}}) \cup \{ (\text{prd}, i) \}$ 

```

Figure 3.4: Algorithm to construct product sets for each k_{ij} .

```

perform global-collect operation on local d vectors
to collect the global d vector dg
for  $i \in \text{mycols}(\mathbf{K})$  do
  for  $j \in \text{Struct}(\mathbf{K}, i)$  and  $j \geq i$  do
     $k_{ji} = 0.0$ 
    for  $(\text{prd}, \text{idx}) \in \text{prds}(k_{ji})$  do
       $k_{ji} = k_{ji} + \text{prd} \times \mathbf{d}_g(\text{idx})$ 

```

Figure 3.5: Node algorithm for the matrix-matrix product $\mathbf{A}\mathbf{D}\mathbf{A}^T$.

array in the algorithm is used to denote the permutation vector that makes this reordering.

In every iteration the actual value of a nonzero k_{ij} will be calculated as (Fig. 3.5)

$$k_{ij} = \sum_{(\text{prd}, \text{idx}) \in \text{prds}(k_{ij})} \text{prd} * d_{\text{idx}}.$$

In this formulation, since only \mathbf{d} vector is needed by processors, communication cost is very low.

3.4 Scalar Operations

Most of the scalar operations can be done on processors without any communication. But some scalar values are needed by every processor. So there will

be duplication of some operations on every processor.

One of the scalar operations needed by PCIPA is the search for a global minimum of some scalar values contained in processors. Most of the parallel architectures have standard routines to do this operation.

3.5 Sparse Linear System Solution

A wealth of solution methods for solving large sparse systems of linear equations has been developed, most of them falling under two categories:

1. *Direct methods* involve the factorization of the system coefficient matrix, usually obtained through Gaussian elimination. Implementations of methods in this class require the use of specific data structures and special pivoting strategies in an attempt to reduce the amount of *fill-in* during Gaussian elimination.
2. *Iterative methods* generate a sequence of approximate solutions to the system of linear equations, usually involving only matrix-vector multiplications in the computation of each iterate. Methods like Jacobi, Gauss-Seidel, Chebychev, Lanczos and the conjugate gradient are attractive by virtue of their low storage requirements, displaying, however, slow convergence unless an effective preconditioner is applied.

The relative merits of each approach depends on such factors as the characteristics of the problem and the host machine. Size, density, nonzero pattern, range of coefficients, structure of eigenvalues and desired accuracy of the solution are some of the problem attributes to be considered. Beyond simple characteristics as speed and size of memory, other aspects of the host machine's architecture play a decisive role both in the selection of a solution method and in specific implementation details. Recent research in sparse matrix techniques concentrate on specializing algorithms that can achieve the most benefit from parallelism, pipelining or vectorization. Also important in the comparison of the two approaches in implementations dedicated to a specific machine is the data transfer rates between various memory media, like disk, main memory, cache memory and register files.

As stated earlier, in this work we make use of Cholesky factorization, which is a direct method for the solution of linear systems. In Chapter 4 we will discuss in detail parallelization of Cholesky factorization and review existing and proposed algorithms. We will also present programming techniques used in the implementation of this step in PLOP.

Chapter 4

Parallel Sparse Cholesky Factorization

Consider a system of linear equations

$$\mathbf{K}\mathbf{x} = \mathbf{b},$$

where \mathbf{K} is an $n \times n$ ¹ symmetric positive definite matrix, \mathbf{b} is a known vector, and \mathbf{x} is the unknown solution vector to be computed. One way to solve the linear system is first to compute the Cholesky factorization

$$\mathbf{K} = \mathbf{L}\mathbf{L}^T,$$

where the Cholesky factor \mathbf{L} is a lower triangular matrix with positive diagonal elements. Then the solution vector \mathbf{x} can be computed by successive forward and backward substitutions to solve the triangular systems

$$\mathbf{L}\mathbf{y} = \mathbf{b}, \mathbf{L}^T\mathbf{x} = \mathbf{y}.$$

4.1 Sequential Algorithms

If \mathbf{K} is a sparse matrix, meaning that most of its entries are zero, then during the course of the factorization some entries that are initially zero in the lower triangle of \mathbf{K} may become nonzero entries in \mathbf{L} . These entries of \mathbf{L} are known as *fill* or *fill-in*. Usually, however, many zero entries in the lower triangle of \mathbf{K} remain zero in \mathbf{L} . For efficient use of computer memory and processing time, it is desirable for the amount of fill to be small, and to store and operate on

¹Though \mathbf{K} appeared as an $m \times m$ matrix in the previous chapters, in this chapter we assume its size as $n \times n$ which is the convention in the literature.

only the nonzero entries of \mathbf{K} and \mathbf{L} .

It is well known that row or column interchanges are not required to maintain numerical stability in the factorization process when \mathbf{K} is positive definite. Furthermore, when roundoff errors are ignored, a given linear system yields the same solution regardless of the particular order in which the equations and unknowns are numbered. This freedom in choosing the ordering can be exploited to enhance the preservation of sparsity in the Cholesky factorization process. More precisely, let \mathbf{P} be any permutation matrix. Since \mathbf{PKP}^T is also a symmetric positive definite matrix, we can choose \mathbf{P} based solely on sparsity considerations. That is, we can often choose \mathbf{P} so that the Cholesky factor $\bar{\mathbf{L}}$ of \mathbf{PKP}^T has less fill than \mathbf{L} . The permuted system is equally useful for solving the original linear system, with the triangular solution phase simply becoming

$$\bar{\mathbf{L}}\mathbf{y} = \mathbf{P}\mathbf{b}, \quad \bar{\mathbf{L}}^T\mathbf{z} = \mathbf{y}, \quad \mathbf{x} = \mathbf{P}^T\mathbf{z}.$$

Unfortunately, finding a permutation \mathbf{P} that minimizes fill is a very difficult combinatorial problem.

Since pivoting is not required in the factorization process, once the ordering is known, the precise locations of all fill entries in \mathbf{L} can be predicted in advance, so that a data structure can be set up to accommodate \mathbf{L} before any numeric computation begins. This data structure need not be modified during subsequent computations, which is a distinct advantage in terms of efficiency. The process by which the nonzero structure of \mathbf{L} is determined in advance is called “symbolic factorization.” Thus, the direct solution of $\mathbf{K}\mathbf{x} = \mathbf{b}$ consists of the following sequence of four distinct steps:

1. *Ordering.* Find a good ordering \mathbf{P} for \mathbf{K} ; that is, determine a permutation matrix \mathbf{P} so that the Cholesky factor \mathbf{L} of \mathbf{PKP}^T suffers little fill.
2. *Symbolic factorization.* Determine the structure of \mathbf{L} and set up a data structure in which to store \mathbf{K} and compute the nonzero entries of \mathbf{L} .
3. *Numeric factorization.* Insert the nonzero entries of \mathbf{K} into the data structure and compute the Cholesky factor \mathbf{L} of \mathbf{PKP}^T .
4. *Triangular solution.* Solve $\mathbf{L}\mathbf{y} = \mathbf{P}\mathbf{b}$ and $\mathbf{L}^T\mathbf{z} = \mathbf{y}$, and then set $\mathbf{x} = \mathbf{P}^T\mathbf{z}$.

Several software packages for serial computers use this basic approach to solve sparse symmetric positive definite linear systems. Detailed explanations of these steps and exposition of the graph theoretical notions used in sparse linear systems can be found in [13]. We now briefly discuss algorithms and methods for performing each of these steps on sequential machines.

4.1.1 Ordering

Despite its simplicity, the minimum degree algorithm produces reasonably good orderings over a remarkably broad range of problem classes. Another strength is its efficiency: as a result of a number of refinements over several years, current implementations are extremely efficient on most problems. George and Liu [14] review a series of enhancements to implementations of the minimum degree algorithm and demonstrate the consequent reductions in ordering time.

Nonzero structure of \mathbf{ADA}^T matrix for woodw problem can be seen in Fig. 4.1. Factorization of this matrix without ordering results in a very dense matrix. But after ordering, sparsity is mostly preserved as seen in Fig. 4.2.

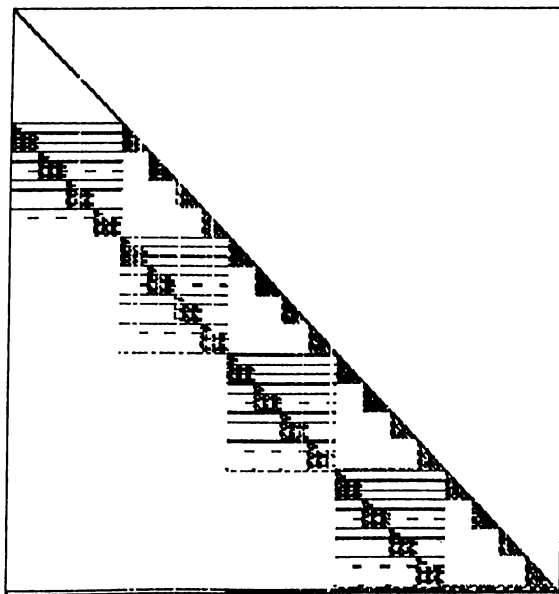


Figure 4.1: Nonzero Entries of \mathbf{ADA}^T matrix for woodw problem.

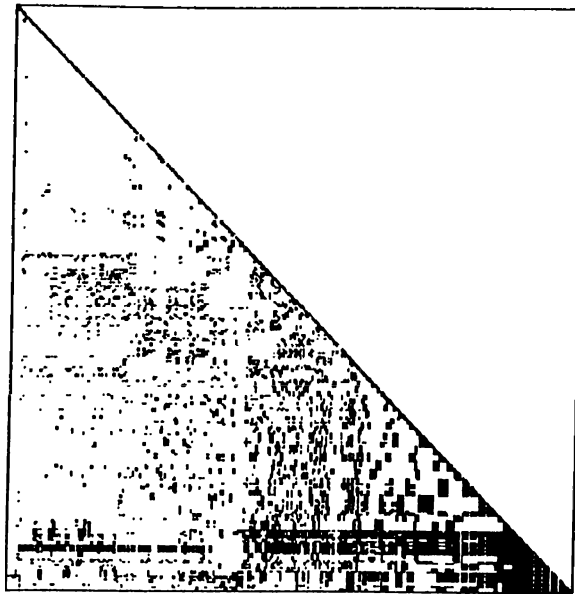


Figure 4.2: Nonzero Entries of ordered \mathbf{L} (factor) matrix for `woodw` problem.

4.1.2 Symbolic Factorization

We use $\text{ColStruct}(\mathbf{M}, k)$ to denote the set of row indices of the nonzero entries in column k of the lower triangular part of the matrix \mathbf{M} . That is,

$$\text{ColStruct}(\mathbf{M}, k) = \{i > k \mid m_{ik} \neq 0\}.$$

Similarly, $\text{RowStruct}(\mathbf{M}, k)$ denotes the set of column indices of the nonzero entries in row k of the lower triangular part of the matrix \mathbf{M} . That is,

$$\text{RowStruct}(\mathbf{M}, k) = \{i < k \mid m_{ki} \neq 0\}.$$

For a given lower triangular Cholesky factor matrix \mathbf{L}^2 , define the function *parent* as follows:

$$\text{parent}(j) = \begin{cases} \min\{i \in \text{ColStruct}(\mathbf{L}, j)\}, & \text{if } \text{ColStruct}(\mathbf{L}, j) \neq \emptyset, \\ j, & \text{otherwise.} \end{cases}$$

Thus, when there is at least one off-diagonal nonzero in column j of \mathbf{L} , $\text{parent}(j)$ is the row index of the first off-diagonal nonzero in that column. It is shown in [13] that

$$\text{ColStruct}(\mathbf{L}, j) \subseteq \text{ColStruct}(\mathbf{L}, \text{parent}(j)) \cup \{\text{parent}(j)\}.$$

²In the subsequent discussion \mathbf{K} and \mathbf{b} will be assumed to be ordered previously, and permutation matrix \mathbf{P} will be implicit.

Moreover, the structure of column j of \mathbf{L} can be characterized as follows:

$$\text{ColStruct}(\mathbf{L}, j) = \text{ColStruct}(\mathbf{K}, j) \cup \left(\bigcup_{i < j, \text{parent}(i)=j} \text{ColStruct}(\mathbf{L}, i) \right) - \{j\}.$$

That is, the structure of column j of \mathbf{L} is given by the structure of the lower triangular portion of column j of \mathbf{K} , together with the structure of each column of \mathbf{L} whose first off-diagonal nonzero is in row j .

This characterization leads directly to an algorithm for performing the symbolic factorization which is already very efficient, with time and space complexity $O(\eta(\mathbf{L}))$, where $\eta(\mathbf{L})$ denotes the number of nonzero entries in \mathbf{L} . An efficient implementation of symbolic factorization algorithm is given in [13]. With its low complexity and an efficient implementation, the symbolic factorization step usually requires less computation than any of the other three steps in solving a symmetric positive definite system by Cholesky factorization.

Once the structure of \mathbf{L} is known, a compact data structure is set up to accommodate all of its nonzero entries. Since only the nonzero entries of the matrix are stored, additional indexing information must be stored to indicate the locations of the nonzero entries.

4.1.3 Numeric Factorization

We will concentrate our attention on two column-oriented methods, column-Cholesky and submatrix-Cholesky. In column-oriented Cholesky factorization algorithms, there are two fundamental types of subtasks:

1. $\text{cmod}(j, k)$: modification of column j by column k , $k < j$,
2. $\text{cdiv}(j)$: division of column j by a scalar.

In terms of these basic operations, high-level descriptions of the column-Cholesky and submatrix-Cholesky algorithms are given in Figs. 4.3 and 4.4.

In column-Cholesky, column j of \mathbf{K} remains unchanged until the index of the outer loop takes on that particular value. At that point the algorithm updates column j with a nonzero multiple of each column $k < j$ of \mathbf{L} for which $l_{jk} \neq 0$. Then l_{jj} is used to scale column j . Column-Cholesky is sometimes said to be a “left-looking” algorithm, since at each stage it accesses needed columns to the left of the current column in the matrix. It is also sometimes referred

```

for  $j = 1$  to  $n$  do
  for  $k \in \text{RowStruct}(\mathbf{L}, j)$  do
     $\text{cmod}(j, k)$ 
   $\text{cdiv}(j)$ 

```

Figure 4.3: Sparse column-Cholesky factorization.

to as a “fan-in” algorithm, since the basic operation is to combine the effects of multiple previous columns on a single subsequent column.

```

for  $k = 1$  to  $n$  do
   $\text{cdiv}(j)$ 
  for  $j \in \text{ColStruct}(\mathbf{L}, k)$  do
     $\text{cmod}(j, k)$ 

```

Figure 4.4: Sparse submatrix-Cholesky factorization.

In submatrix-Cholesky, as soon as column k is completed, its effects on all subsequent columns are computed immediately. Thus, submatrix-Cholesky is sometimes said to be a “right-looking” algorithm, since at each stage columns to the right of the current column are modified. It is also sometimes referred to as a “fan-out” algorithm, since the basic operation is for a single column to affect multiple subsequent columns.

Relevant to the topic of sparse factorization, we introduce the concept of *elimination tree* which is useful in analyzing and efficiently implementing sparse factorization algorithms [17, 20].

The elimination tree $T(\mathbf{K})$ associated with the Cholesky factor \mathbf{L} of a given matrix \mathbf{K} has $\{v_1, v_2, \dots, v_n\}$ as its node set, and has an edge between two vertices v_i and v_j , with $i > j$, if $i = \text{parent}(j)$, where parent is the function defined in Section 4.1.2. In this case, node v_i is said to be the *parent* of node v_j , and node v_j is a *child* of node v_i . The elimination tree is fixed for a given ordering and is a heap ordered tree with v_n as its root. It captures all the dependencies between the columns of \mathbf{K} in the following sense. If there is an edge (v_i, v_j) , $i < j$, in the elimination tree then the factorization of column j can not be completed unless that of column i is completed. Elimination tree

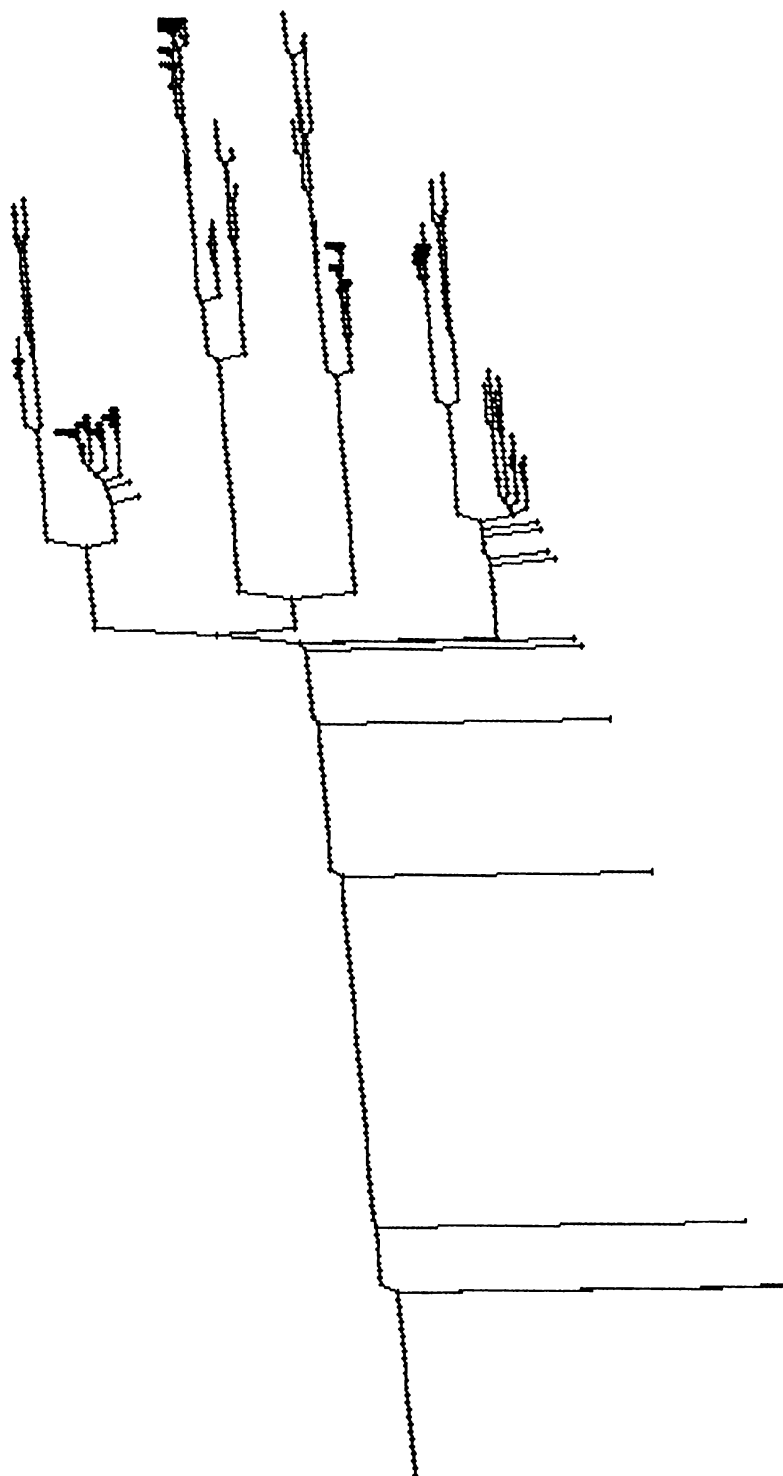


Figure 4.5: Elimination tree for `woodw`.

for the problem `woodw` (with 1098 nodes) is given in Fig. 4.5 which is a typical one for NETLIB problems.

The row structure $\text{RowStruct}(\mathbf{L}, j)$ is a pruned subtree rooted at node v_i in the elimination tree. Let $T(j)$ denote this subtree. It can be shown that column j can be completed only after every column in $T(j)$ has been computed. It also follows that the columns that receive updates from column j are ancestors of j in $T(\mathbf{K})$. In other words, the node set $\text{ColStruct}(\mathbf{L}, j)$ is a subset of the ancestors of j in the tree. Let

$$\text{anc}(j) = \{i \mid v_i \text{ is an ancestor of } v_j \text{ in } T(\mathbf{K})\}.$$

We have,

$$\text{ColStruct}(\mathbf{L}, j) \subseteq \text{anc}(j). \quad (4.1)$$

4.1.4 Triangular Solution

The structure of the forward and backward substitution algorithms is more or less dictated by the sparse data structure used to store the triangular Cholesky factor \mathbf{L} and by the structure of the elimination tree $T(\mathbf{K})$. Because triangular solution requires many fewer floating-point operations than the factorization step that precedes it, the triangular solution step usually requires only a small fraction of the total time to solve a sparse linear system on conventional sequential computers.

The sequential forward substitution algorithm can be stated as follows:

$$y_i = \left(b_i - \sum_{j \in \text{RowStruct}(\mathbf{L}, i)} l_{ij} * y_j \right) / l_{ii}, \quad i = 1, 2, \dots, n.$$

The sequential backward substitution algorithm is:

$$x_i = \left(y_i - \sum_{j \in \text{RowStruct}(\mathbf{L}^T, i)} l_{ij}^T * x_j \right) / l_{ii}^T, \quad i = n, n-1, \dots, 1.$$

4.2 Parallel Algorithms

On parallel machines the same sequence of four distinct steps is performed: ordering, symbolic factorization, numeric factorization, and triangular solution.

However, both shared-memory and distributed-memory parallel computers require an additional step to be performed: the tasks into which the problem is decomposed must be mapped onto the processors. Obviously, one of the goals in mapping the problem onto the processors is to ensure that the work load is balanced across all processors. Moreover, it is desirable to schedule the problem so that the amount of synchronization and/or communication is low.

We now proceed to discuss each of these five steps.

4.2.1 Ordering and Symbolic Factorization

One issue associated with the ordering problem in a parallel environment is the determination of an ordering appropriate for performing the subsequent factorization efficiently on the parallel architecture in question. However, there have been no systematic attempts to develop metrics for measuring the quality of parallel orderings. Thus far, most work on the parallel ordering problem has used elimination tree height as the criterion for comparing orderings, with short trees assumed to be superior to taller trees, but with little more than intuition as a basis for this choice.

A separate problem is the need to compute the ordering in parallel on the same machine on which the other steps of the solution process are to be performed. The ordering algorithm discussed earlier, namely, minimum degree is extremely efficient and normally constitute only a small fraction of the total execution time in solving a sparse system.

The sequential algorithm for computing the symbolic factorization is remarkably efficient, and so once again we find ourselves with little work to distribute among the processors, so that good efficiency is difficult to achieve in a parallel implementation.

To summarize, the problem of computing effective parallel orderings is very difficult and remains largely untouched by research efforts to date. Furthermore, primary concern of this work, interior point algorithms, requires only one ordering and symbolic factorization step to be used on several subsequent iterations. So we perform ordering and symbolic factorization sequentially in the initialization phase of the parallel algorithm (PCIPA).

4.2.2 Task Partitioning and Scheduling

In this section we will address the problem of mapping the computational work in Cholesky factorization on distributed-memory message-passing parallel computers. On these machines, the lack of globally accessible memory means that issues concerned with data locality are dominant considerations. Currently, there is no efficient means of implementing dynamic load balancing on these machines for problems of this type. Thus, a static assignment of tasks to processors is normally employed in this setting, and such a mapping must be determined in advance of the factorization, based on the tradeoffs between load balancing and the cost of interprocessor communication. We map the columns of \mathbf{K} among the processors rather than the individual elements of the matrix because this level of granularity is well suited for most of the multiprocessors commercially available today.

The elimination tree contains information on data dependencies among tasks and the corresponding communication requirements. Thus, the elimination tree is an extremely helpful guide in determining an effective assignment of columns (and corresponding tasks) to processors. A graphical interpretation of the factorization can be obtained using the elimination tree. Computing a column of the Cholesky factor corresponds to removing or eliminating that node from the tree. For example, at the first step, any or all of the leaf nodes can be eliminated. Moreover, they can be eliminated simultaneously, if enough processors are available. This creates a new set of leaf nodes in the tree, which can now be eliminated, and so on.

In general the leaf nodes in the elimination tree denote all the independent columns of the sparse matrix, and the paths down the elimination tree to the root specify the column dependencies. Note that it is the column dependencies that give rise to communication or synchronization, since computing column i of \mathbf{L} will require other columns on which column i depends. Thus, the elimination tree provides precise information about the column dependencies in computing \mathbf{L} and hence can be used to assign the columns of the sparse matrix to different processors.

After the elimination tree has been generated, the next step is to use it in mapping the columns onto the processors.


```

for  $i = 1$  to  $n$  do
     $fcnt(i) = 0$ 
for  $i = 1$  to  $n$  do
     $fcnt(i) = fcnt(i) + nz(i);$ 
    for  $k \in ColStruct(\mathbf{L}, i)$  do
         $fcnt(k) = fcnt(k) + nz(i)$ 
         $nz(i) = nz(i) - 1$ 

```

Figure 4.6: Algorithm for predicting the number of floating-point operations required to generate each column of \mathbf{L} .

In the early work on this problem, successive levels in the elimination tree were wrap-mapped to the processors using the following scheme: column i is assigned to processor $(i - 1) \bmod \mathcal{P}$. This results in good load balancing for the problem, but it also often results in unnecessarily high message volume. The “subtree-to-subcube” mapping [12], does an excellent job of reducing communication while maintaining good load balance for model grid problems and other problems with similar regularity in their structure. It is difficult, however to use the subtree-to-subcube mapping for more irregular problems.

The mapping scheme presented by Geist and Ng [10] can be thought of as a generalization of the subtree-to-subcube mapping scheme to arbitrarily unbalanced elimination trees. Their algorithm is described below.

Given the structure of \mathbf{K} , the number of nonzero entries in each column of \mathbf{L} , and the elimination tree, it is possible to calculate the number of floating-point operations required in computing each column of \mathbf{L} . Since we perform ordering and symbolic factorization steps prior to scheduling in PLOP, the structure of \mathbf{L} is readily available in our case. So our algorithm which is shown in Fig. 4.6 is simpler than Geist and Ng’s which is an extension of the algorithm for computing the amount of fill. The vector nz initially contains the number of nonzero entries in each column of \mathbf{L} . This vector is destroyed during the generation of the vector $fcnt$, which contains the number of multiplications and divisions performed on each column of \mathbf{L} . These operation counts will be used as nodal weights.

Given an arbitrary tree and \mathcal{P} processors, our task is to find the smallest set of branches in the tree such that this set can be partitioned into exactly

```

for  $i = 1$  to  $n$  do
     $\text{nodewt}(i) = 0$ 
for  $i = 1$  to  $n$  do
     $\text{nodewt}(i) = \text{nodewt}(i) + \text{fcnt}(i);$ 
    if  $\text{parent}(i) \neq i$ 
         $\text{nodewt}(\text{parent}(i)) = \text{nodewt}(\text{parent}(i)) + \text{nodewt}(i)$ 

```

Figure 4.7: Algorithm to generate tree weights.

\mathcal{P} subsets, all of which require approximately the same amount of work. Over this class of solutions we wish to maximize the operation counts in the set of branches.

Geist and Ng's strategy involves a breath first search of the elimination tree, cutting off branches and applying a heuristic bin packing algorithm to the set of branches. The procedure is applied iteratively until the work load across all processors meets a user defined tolerance. That is, the iterative procedure will be terminated when the difference in work load between any two processors is less than a user specified parameter. Each processor is then assigned the set of branches in a particular bin. The remaining nodes are assigned to all processors in a wrap around manner as described earlier.

The key to this strategy is knowing how large each of the branches is without searching down it each time. The relative size of the branches determines which parts of the tree need pruning and which parts should be taken as a whole. This is accomplished by using a weighted elimination tree. Each node i of the tree is given a weight $\text{nodewt}(i)$, which is equal to the sum of the weights of its children and the number of floating-point operations performed on column i of \mathbf{L} . The algorithm in Fig. 4.7 describes how these weights are generated. The final task scheduling algorithm is shown in Figs. 4.8 and 4.9.

Each column l_{*k} is stored on one and only one of \mathcal{P} available processors. An n -vector map is required to record the distribution of columns to processors: if column k is stored on processor p , then $\text{map}(\mathbf{L}, k) = p$. We use $\text{mycols}(\mathbf{L})$ to denote the set of columns owned by a processor.

```

ratio = 0
ip =  $\mathcal{P}$  - 1
while ( ratio  $\leq$  tolerance )
    ip = ip + 1
    partition tree into at least ip branches
    initialize all bins to 0
    while (list-of-branches not empty)
        find branch with max weight in list-of-branches
        add branch to minimum bin
        delete branch from list-of-branches
    find max bin and min bin
    ratio = minbin / maxbin
assign nodes in bins to processors
assign rest nodes in wrapped manner

```

Figure 4.8: Task Scheduling Algorithm.

```

if ( first call )
    initialize list as first ip branches of tree
while ( size-of-list < ip )
    find node in list with max nodewt
    find next branch in subtree with root node
    add next branch and all its siblings to list
    delete node from the list
return list-of-branches

```

Figure 4.9: Algorithm to partition the tree.

4.2.3 Numeric Factorization

Algorithms for distributed-memory machines are usually structured around some prior distribution of the data to the processors. In order to keep the cost of interprocessor communication at acceptable levels, it is essential for the algorithm to make local use of local data as much as possible. The distributed fan-out and fan-in algorithms are typical examples of this type of distributed algorithm. Both of them use the column assignment to distribute among the processors the tasks found in the outer loop of one of the serial implementations of sparse Cholesky factorization of Section 4.1.3.

A straightforward parallelization of the submatrix-Cholesky algorithm shown in Fig. 4.4 results in the *fan-out* algorithm which was introduced in [11]. The outer (k) loop is distributed among the processors using $\text{map}(\mathbf{L}, \cdot)$. After each owned column k is normalized, it is sent out to processors which own columns updated by it. The whole process is structured in a data-driven form, where the arrival of a source column triggers the local actions of updating owned targets using it. The algorithm is terminated on each processor when all of its owned columns have been normalized and sent out to others.

Due to several weaknesses it has been superseded by fan-in algorithms. The distributed fan-out algorithm incurs greater interprocessor communication costs than the fan-in algorithm, both in terms of total number of messages and total message volume. It simply does not exploit a good communication-reducing column mapping, such as the one discussed in Section 4.2.2.

The fan-in algorithm for parallel sparse factorization [3] is motivated by a desire to reduce the amount of communication required. In the fan-out algorithm, for every source-target pair involving an update, the processor which owns the source column sends it to the processor owning the target column. In general, if a particular processor owns more than one of the source columns updating a particular target column, each of the source columns must be explicitly transmitted. Since the multiplier that is used when actually performing the update on the target is obtained from the source column, the contribution of that source can be computed at the source processor itself, and then sent to the target processor. A single message combining all such contributions from

```

for  $j = 1$  to  $n$  do
  if  $j \in \text{mycols}(\mathbf{L})$  or  $\text{RowStruct}(\mathbf{L}, j) \cap \text{mycols}(\mathbf{L}) \neq \emptyset$  do
     $\mathbf{u} = 0$ 
    for  $k \in \text{RowStruct}(\mathbf{L}, j) \cap \text{mycols}(\mathbf{L})$  do
       $\mathbf{u} = \mathbf{u} + \mathbf{u}(j, k)$ 
    if  $j \notin \text{mycols}(\mathbf{L})$  do
      send  $\mathbf{u}$  to processor  $\text{map}(\mathbf{L}, j)$ 
    else
      incorporate  $\mathbf{u}$  into the factor column  $j$ 
      while ( any aggregated update column for column  $j$ 
        remains unreceived ) do
        receive in  $\mathbf{u}$  another aggregated update for column  $j$ 
        incorporate  $\mathbf{u}$  into the factor column  $j$ 
      cdiv( $j$ )

```

Figure 4.10: Fan-in Cholesky factorization algorithm.

the source processor for a particular target column can be formed and this “aggregate” column transmitted instead of each of the source columns. Instead of the previously considered submatrix-Cholesky, by using column-Cholesky (Fig. 4.3), each target will be considered in the outer loop, enabling the formation of the combined contribution to it all at once, instead of in different iterations of the outer loop, so that the space used for storing the combined contributions can be reused for each target. Fig. 4.10 gives a pseudocode for the resulting fan-in algorithm. The task scheduling algorithm of Section 4.2.2 can be used for mapping the columns to processors. However, the amount of work at each node of the elimination tree is now considered to be the sum of the number of update operations from the node and the number of normalization operations for the node.

A visual picture of the communication pattern of the fan-in algorithm is given in Fig. 4.11. The spacetime figure illustrates snapshots of the execution of the algorithm on an iPSC/2 hypercube, with time on the horizontal axis. Processor activity is shown by horizontal lines and interprocessor communication by slanted lines. The horizontal line corresponding to each processor is either solid or blank, depending on whether the processor is busy or idle, respectively. Each message sent between processors is shown by a line drawn from the sending processor at the time of transmission to the receiving processor at the time of reception of the message.

Utilization count diagram given in Fig. 4.11 shows the total number of

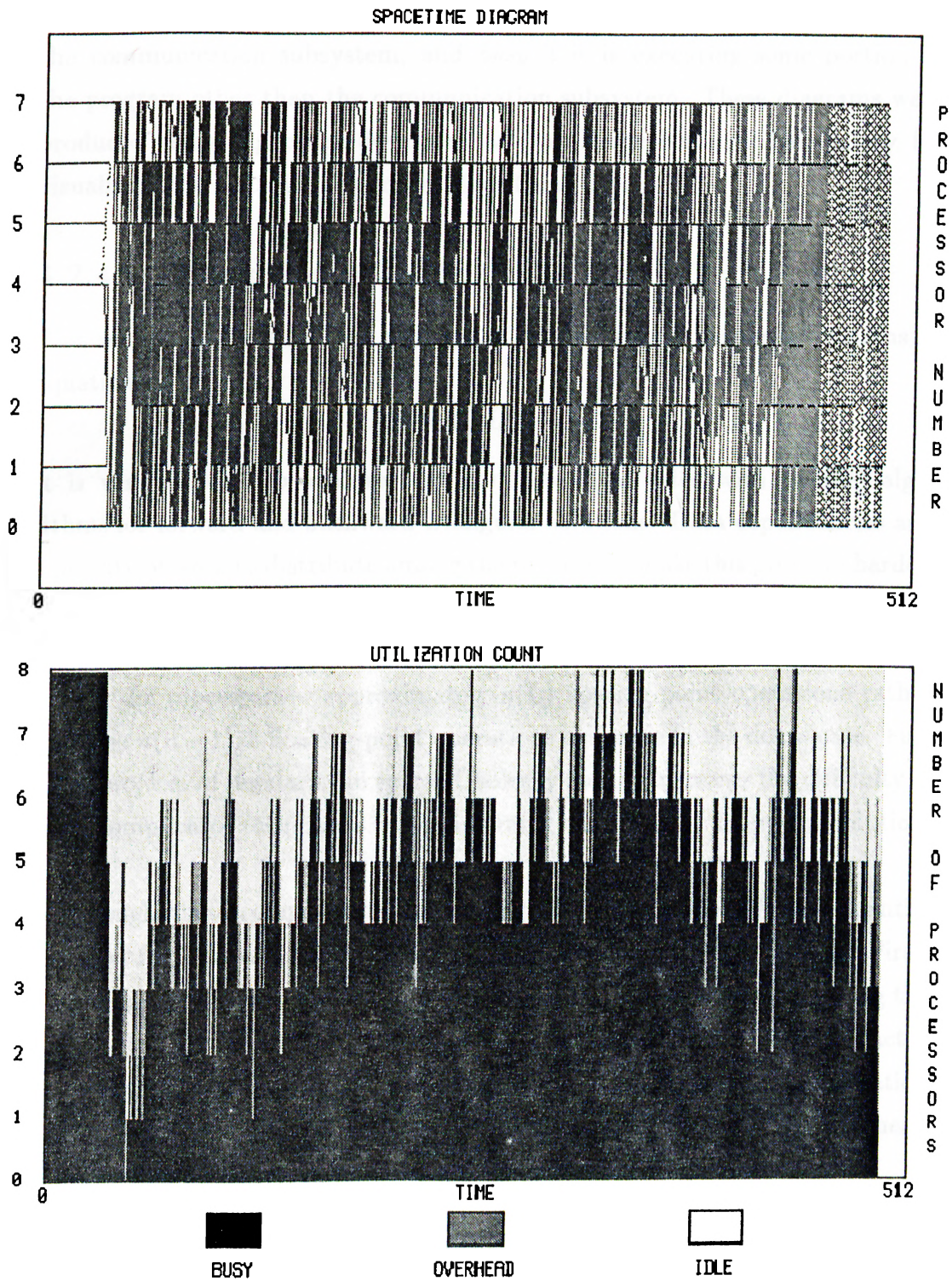


Figure 4.11: Spacetime diagram and utilization count for fan-in factorization of 80bau3b problem.

processors in each of three states — busy, overhead, and idle — as a function of time. Each processor is categorized as *idle* if it has suspended execution awaiting a message that has not yet arrived, *overhead* if it is executing in the communication subsystem, and *busy* if it is executing some portion of the program other than the communication subsystem. These diagrams were produced using a package developed at Oak Ridge National Laboratory for visualizing the behavior of parallel algorithms [16].

4.2.4 Triangular Solution

This section will address the problem of solving sparse triangular systems of equations:

$$\mathbf{L}\mathbf{y} = \mathbf{b}, \mathbf{L}^T\mathbf{x} = \mathbf{y}.$$

It is very difficult to achieve high computational rates with parallel algorithms for forward and backward triangular solutions. Data dependencies and a paucity of work to distribute among the processors make this problem harder, even for the dense case. When solving the sparse problems, due to preservation of sparsity in the factor matrix, there is usually far less work to distribute among the processors— approximately $\eta(\mathbf{L})$ floating-point operations rather than the $n(n-1)/2$ floating-point operations available in the dense case. Furthermore, loss of regularity in sparse Cholesky factors increases the difficulty of using complicated techniques (*e.g.*, pipelining) to speed up triangular solution.

Though these computations take much less time compared to sequential Cholesky factorization, there are several reasons to solve them in parallel. First, the triangular systems may need to be solved a large number of times using the same factor matrix but using different \mathbf{b} vectors. Second, the Cholesky factor is usually available across the processors as a result of parallel factorization and in such a case parallel solution of triangular systems avoids the overhead of collecting the factor matrix into a single processor.

Elimination Tree Based FS (EBFS)

Kumar *et al.* [20] exploit the elimination tree concept to develop forward and backward triangular solvers. They consider the situation when the number of processors is same as the number of columns of \mathbf{L} , and column i along with b_i are assigned to processor i . Their parallel algorithm for FS phase (EBFS),

```

    k = mynode()
    u = 0
    if vk is not a leaf
        while smod(k) > 0
            receive update vector u' from a child vi
            for j ∈ Struct(u') do
                uj = uj + u'j
            decrement smod(k)
    yk = (bk - uk) / lkk
    for j ∈ ColStruct(L, k) do
        uj = uj + ljk * yk
    send u(j | j ∈ ColStruct(L, k)) to map(L, parent(k))

```

Figure 4.12: Node algorithm for EBFS with $\mathcal{P} = n$.

starts from the leaves of the elimination tree. An update vector \mathbf{u} of size n is associated with each processor. \mathbf{u} is initialized to 0. The processor containing a leaf node v_i computes the value of y_i as

$$y_i \leftarrow b_i / l_{ii}.$$

It then modifies its update vector as follows:

$$\forall j \in \text{ColStruct}(\mathbf{L}, i), u_j \leftarrow l_{ji} y_i.$$

The modified update vector is sent to the parent of v_i . However, instead of sending the entire update vector, only the elements in \mathbf{u} corresponding to the i -th column structure are sent to the parent of v_i . An internal vertex v_i waits till it receives update vectors from all of its children and adds each of them to its update vector. The value of y_i is calculated as

$$y_i \leftarrow (b_i - u_i) / l_{ii}.$$

The update vector is then modified as follows:

$$\forall j \in \text{ColStruct}(\mathbf{L}, i), u_j \leftarrow u_j + l_{ji} y_i.$$

The modified update vector is sent to the parent of v_i . Here also, only the relevant elements of the update vector are sent. When the computation terminates, the processor i contains the value y_i . The algorithm is given in Fig. 4.12. In the algorithm $\text{smod}(k)$ initially contains the number of children of v_k . Space-time diagram and utilization count for EBFS are shown in Fig. 4.13 where 8 processors are used to solve 80bau3b problem.

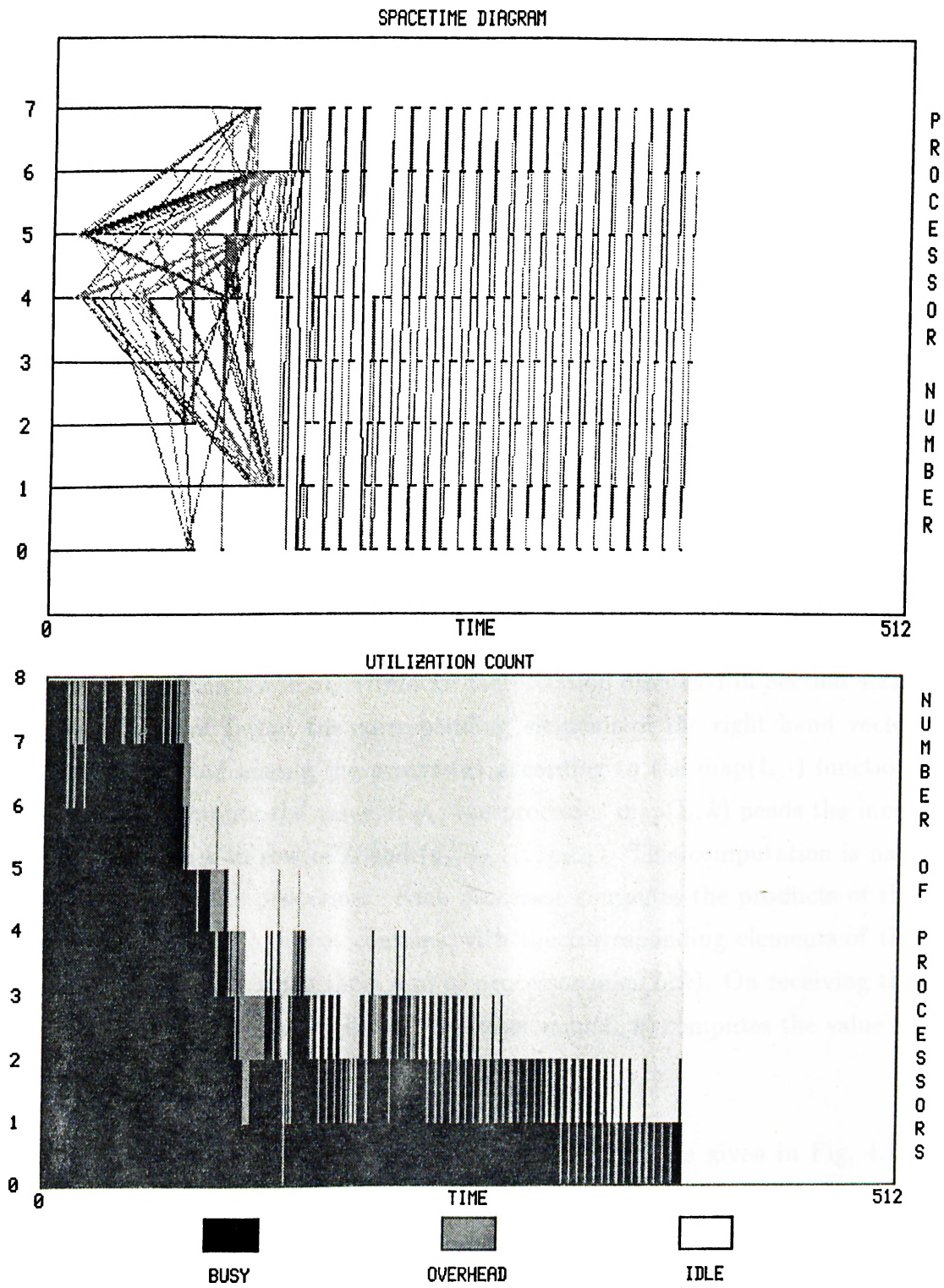


Figure 4.13: Spacetime diagram and utilization count for EBFS on 80bau3b problem.

```

for  $j = 1$  to  $n$  do
  if  $j \in \text{mycols}(\mathbf{L})$  or  $\text{RowStruct}(\mathbf{L}, j) \cap \text{mycols}(\mathbf{L}) \neq \emptyset$  do
     $u = 0$ 
    for  $k \in \text{RowStruct}(\mathbf{L}, j) \cap \text{mycols}(\mathbf{L})$  do
       $u = u + y_k * l_{jk}$ 
    if  $j \notin \text{mycols}(\mathbf{L})$  do
      send  $u$  to processor  $\text{map}(\mathbf{L}, j)$ 
    else
      while any aggregated update for  $y_j$  remains unreceived do
        add to  $u$  another aggregated update for  $y_j$ 
       $y_j = (b_j - u) / l_{jj}$ 

```

Figure 4.14: Fan-in FS algorithm (FIFS).

Kumar *et al.* later consider the case where the columns are mapped to processors according to $\text{map}(\cdot)$ function and modify the algorithm to remove the redundancy in the messages arising in this situation (see [20]).

Fan-in FS (FIFS)

The sparse forward solution algorithm proposed by George *et al.* [12] is an adaptation of the fan-in algorithm for factorization discussed in Section 4.2.3. The columns of \mathbf{L} and the corresponding elements of the right hand vector \mathbf{b} are distributed among the processors according to the $\text{map}(\mathbf{L}, \cdot)$ function. In order to compute the value of y_k , the processor $\text{map}(\mathbf{L}, k)$ needs the inner product of the k -th row of \mathbf{L} and $(y_1, y_2, \dots, y_{k-1})$. This computation is partitioned among the processors. Each processor computes the products of the elements of the k -th row it contains with the corresponding elements of the solution vector and sends their sum to processor $\text{map}(\mathbf{L}, k)$. On receiving the contributions of all the processors, processor $\text{map}(\mathbf{L}, k)$ computes the value of y_k .

Spacetime diagram and utilization count for FIFS are given in Fig. 4.15 which has the same time scale with Fig. 4.13 given for EBFS. These figures reveal that elimination tree based algorithm has apparently low message count and it takes less time to complete the FS in this setting.

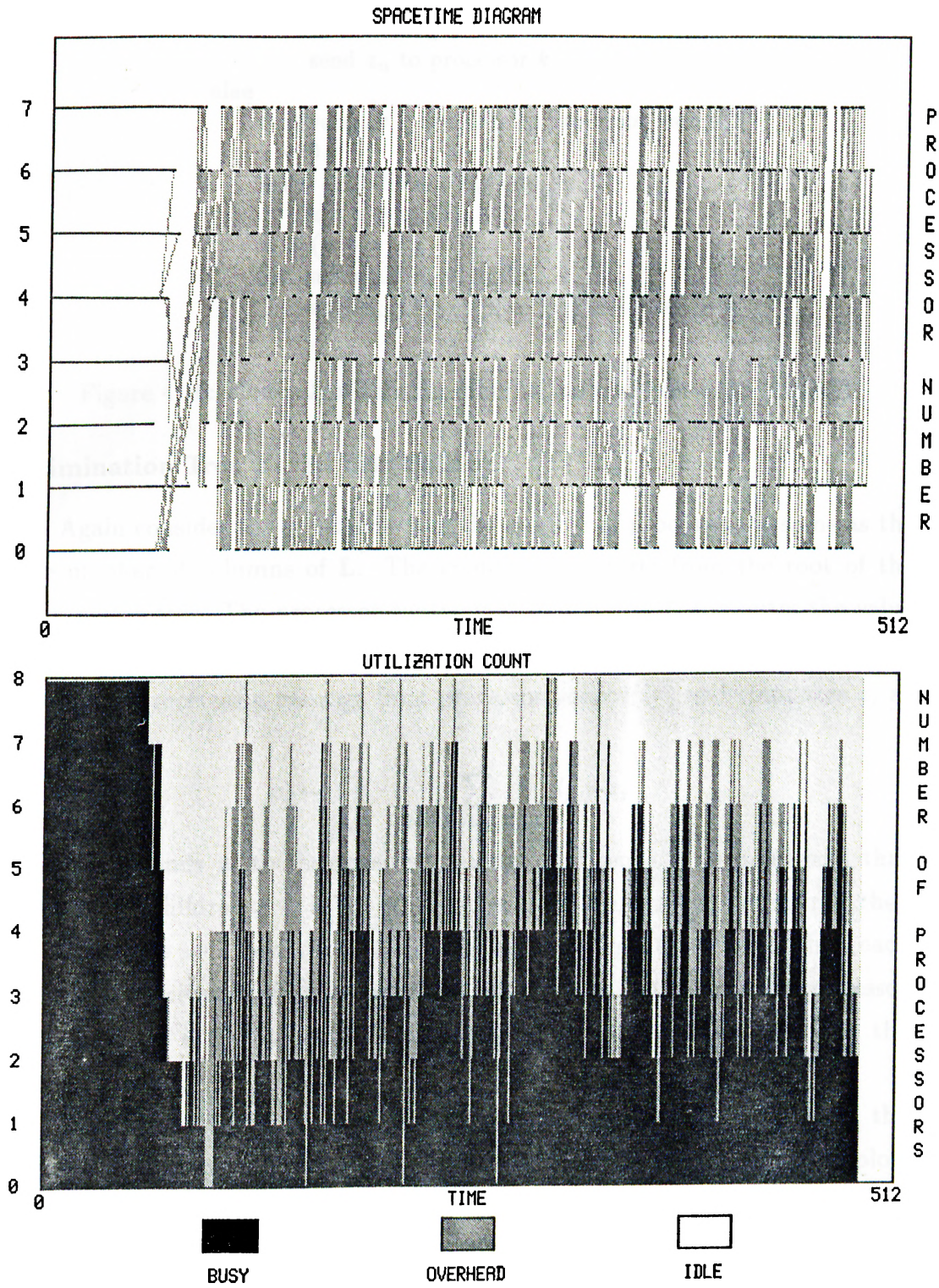


Figure 4.15: Spacetime diagram and utilization count for fan-in FS on 80bau3b problem.

```

i = mynode()
if (i = n)
     $x_n = y_n / l_{nn}$ 
    for each child  $v_k$  of  $v_n$  do
        send  $x_n$  to processor  $k$ 
else
    receive  $x_k, k \in \text{anc}(i)$  from processor  $\text{parent}(i)$ 
     $u = 0$ 
    for  $k \in \text{ColStruct}(\mathbf{L}, i)$  do
         $u = u + x_k * l_{ki}$ 
     $x_i = (y_i - u) / l_{ii}$ 
    for each child  $v_k$  of  $v_i$  do
        send  $x_j, j \in \{i\} \cup \{\text{anc}(i)\}$  to processor  $k$ 

```

Figure 4.16: Node algorithm for elimination tree based BS (EBBS).

Elimination Tree Based BS (EBBS)

Again consider the situation when the number of processors is same as the the number of columns of \mathbf{L} . The computation starts from the root of the elimination tree. The processor containing column n of \mathbf{L} computes the value of $x_n \leftarrow y_n / l_{nn}$. It sends x_n to processors that contain children of v_n . Each processor i receives a message from processor $\text{parent}(i)$, and computes x_i as follows:

$$x_i \leftarrow \left(y_i - \sum_{j \in \text{ColStruct}(\mathbf{L}, i)} l_{ji} * x_j \right).$$

It then appends x_i to the received message and sends it to processors that contain the children of v_i in the elimination tree. If v_i is a leaf in the tree, then no message is sent from processor i . During the course of the algorithm, each processor receives $x_k, k \in \text{anc}(i)$ which ensures, by (4.1) that each processor receives $x_k, k \in \text{ColStruct}(\mathbf{L}, i)$. Hence, the algorithm terminates with the correct results.

Again, as in EBFS, Kumar *et al.* [20] later consider the case where the columns are mapped to processors according to $\text{map}(\cdot)$ function and exploit $\text{map}(\cdot)$ to reduce the size of the messages.

Send-Forward BS (SFBS)

The sparse backward solution discussed in [12] is based on the dense backward algorithm in which the value of x_k is broadcast to all the processors as soon as it is computed. All the processors update elements of \mathbf{y} vector in

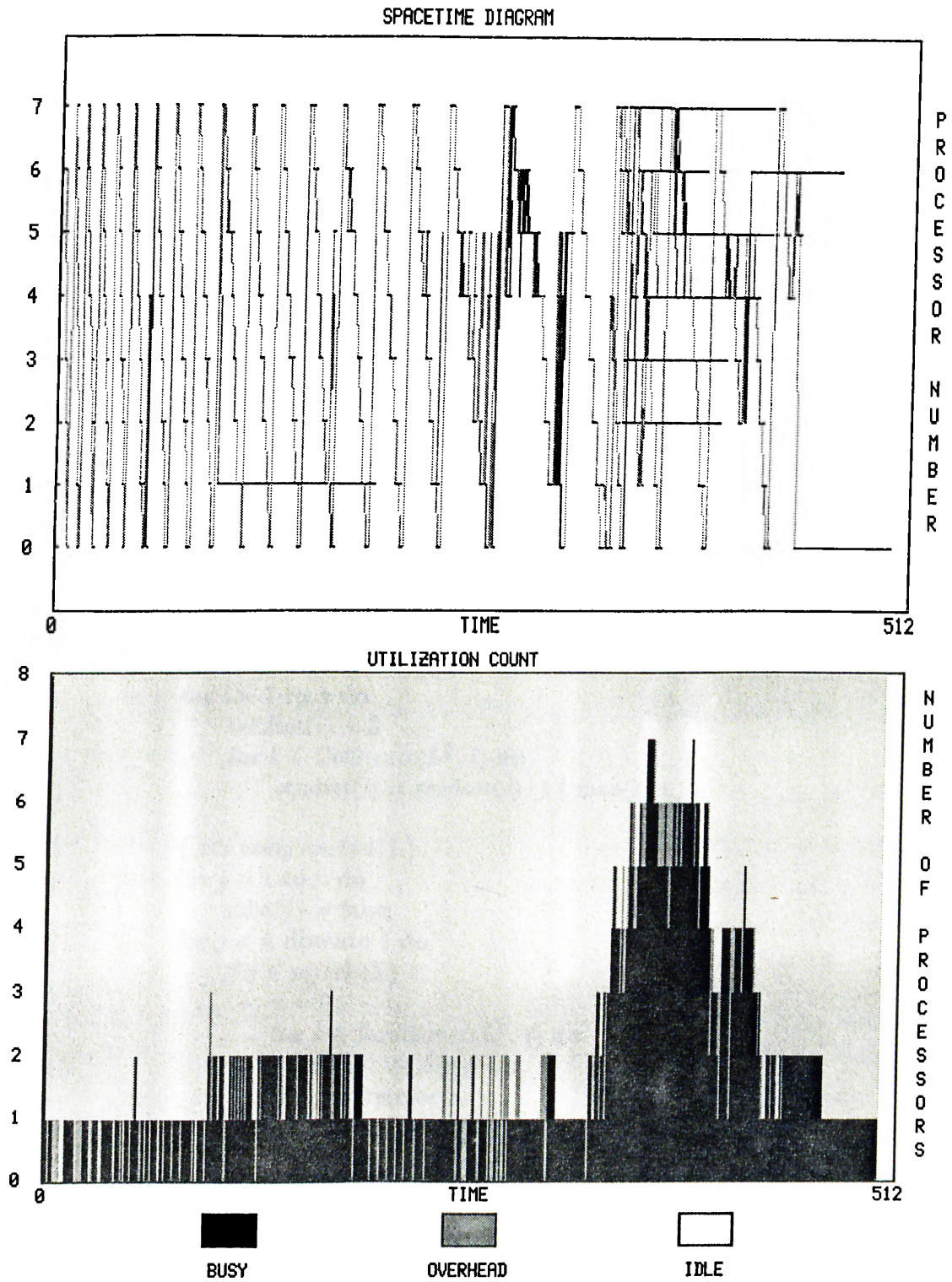


Figure 4.17: Spacetime diagram and utilization count for EBBS on 80bau3b problem.

```

for  $j = 1$  to  $n$  do
     $\text{valid}(j) = \text{false}$ 
for  $j = n$  downto  $1$  do
    if  $j \in \text{mycols}(\mathbf{L})$ 
         $u = 0$ 
        for  $k \in \text{RowStruct}(\mathbf{L}^T, j)$  do
            if not  $\text{valid}(k)$ 
                receive  $x_k$ 
                 $\text{valid}(k) = \text{true}$ 
             $u = u + x_k * l_{kj}$ 
         $x_j = (y_j - u) / l_{jj}$ 
         $\text{valid}(j) = \text{true}$ 
        broadcast  $x_j$  to all processors

```

Figure 4.18: Broadcast BS algorithm.

```

{ initialize the  $\text{sendset}()$  }
for  $i = 1$  to  $n$  do
     $\text{sendset}(i) = \emptyset$ 
    for  $k \in \text{ColStruct}(\mathbf{L}^T, i)$  do
         $\text{sendset}(i) = \text{sendset}(i) \cup \{ \text{map}(\mathbf{L}, k) \}$ 

{ BS using  $\text{sendset}()$  }
for  $j = 1$  to  $n$  do
     $\text{valid}(j) = \text{false};$ 
for  $j = n$  downto  $1$  do
    if  $j \in \text{mycols}(\mathbf{L})$ 
         $u = 0$ 
        for  $k \in \text{RowStruct}(\mathbf{L}^T, j)$  do
            if not  $\text{valid}(k)$ 
                receive  $x_k$ 
                 $\text{valid}(k) = \text{true}$ 
             $u = u + x_k * l_{kj}$ 
         $x_j = (y_j - u) / l_{jj}$ 
         $\text{valid}(j) = \text{true}$ 
        for  $i \in \text{sendset}(j)$  do
            send  $x_j$  to processor  $i$ 

```

Figure 4.19: Send-forward BS (SFBS) algorithm.

parallel as follows: $y_j \leftarrow y_j - l_{jk}x_k$, $j < k$. Next, the processor containing the $(k-1)$ th row of \mathbf{L}^T computes x_{k-1} and broadcasts the same, and so on. Broadcast BS (BCBS) algorithm is shown in figure 4.18.

In the sparse case, we can avoid broadcasting x_k , if we know the structure of the k -th column of \mathbf{L}^T . We only need to send x_k to all the processors which contain rows of \mathbf{L}^T that have nonzero entries in column k . But, since \mathbf{L}^T is stored row-wise, the column structure is not readily available. In PLOP this problem is solved in the preprocessing phase. After finding nonzero structure of \mathbf{L}^T column-wise (ColStruct), a data structure is formed such that, $\text{sendset}(k)$ contains the processors which need x_k . This new algorithm which we name send-forward BS (SFBS) is shown in Fig. 4.19.

Buffered BS

Send-forward BS algorithm can be improved radically. In SFBS the value of x_k is sent to processors in $\text{sendset}(k)$ as soon as it is computed. This process causes the sending processor to do early work and to delay the necessary work, because most of the processors in $\text{sendset}(k)$ need x_k only many iterations later. Furthermore, sending x_k values individually incurs a high message count. So, if we buffer x values that will be sent to a processor and in subsequent iterations send them in a combined message, we will reduce the number of messages and prevent the early sends. These ideas are used to develop the buffered BS algorithm (BFBS) given in Fig. 4.20.

Here, $\text{sendset}(k)$ is modified to contain the tuples (s, t) such that processor s contains rows of \mathbf{L}^T that have non-zeros in column k and t is the highest index among these columns. In other words, the tuple (s, t) means that the earliest iteration in which x_k is needed by processor s is t .

Additionally, we introduce buffers where $\text{buf}(i)$ is used to combine x_k values to be sent to processor i . Associated with $\text{buf}(i)$, there is a value $\text{maxj}(i)$ which denotes the earliest iteration in which the current contents of $\text{buf}(i)$ is to be sent.

In the k -th iteration of the algorithm, the value of x_k is computed and added

to buffers denoted in $\text{sendset}(k)$ by the processor having column k . Other processors, control the buffer $\text{buf}(\text{map}(\mathbf{L}, k))$ and if the current iteration value exceeds $\text{maxj}(\text{map}(\mathbf{L}, k))$, the buffer is sent to processor $\text{map}(\mathbf{L}, k)$.

Spacetime diagram and utilization count for BFBS are given in Fig. 4.21 which has the same time scale with Fig. 4.17 given for EBBS. Though elimination tree based algorithm has apparently low message count it takes much more time than BFBS because of idle waitings.


```

{ initialize the sendset() }
for  $i = 1$  to  $n$  do
    sendset( $i$ ) =  $\emptyset$ 
    for  $k \in \text{ColStruct}(\mathbf{L}^T, i)$  do
         $p = \text{map}(\mathbf{L}, k)$ 
        if  $\exists (p, t) \in \text{sendset}(i)$ 
            if  $k > t$ 
                sendset( $i$ ) = (sendset( $i$ ) -  $\{(p, t)\}$ )  $\cup$   $\{(p, k)\}$ 
            else
                sendset( $i$ ) = sendset( $i$ )  $\cup$   $\{(p, k)\}$ 

{ BS using sendset() }
for  $j = 1$  to  $n$  do
    valid( $j$ ) = false;
for  $i = 1$  to  $\mathcal{P}$  do
    buf( $i$ ) =  $\emptyset$ 
    maxj( $i$ ) = 0
for  $j = n$  downto 1 do
    if  $j \in \text{mycols}(\mathbf{L})$ 
         $u = 0$ 
        for  $k \in \text{RowStruct}(\mathbf{L}^T, j)$  do
            while not valid( $k$ )
                receive next message  $W$ 
                for  $(v, t) \in W$ 
                     $x_t = v$ 
                    valid( $t$ ) = true
                 $u = u + x_k * l_{kj}$ 
             $x_j = (y_j - u) / l_{jj}$ 
            valid( $j$ ) = true
            for  $(i, t) \in \text{sendset}(j)$  do
                buf( $i$ ) = buf( $i$ )  $\cup$   $\{(x_j, j)\}$ 
                if  $t > \text{maxj}(i)$ 
                    maxj( $i$ ) =  $t$ 
        else
             $i = \text{map}(\mathbf{L}, j)$ 
            if maxj( $i$ )  $\geq j$ 
                send  $(v, t) \in \text{buf}(i)$  to processor  $i$ 
                buf( $i$ ) =  $\emptyset$ 
                maxj( $i$ ) = 0

```

Figure 4.20: Buffered BS algorithm (BFBS).

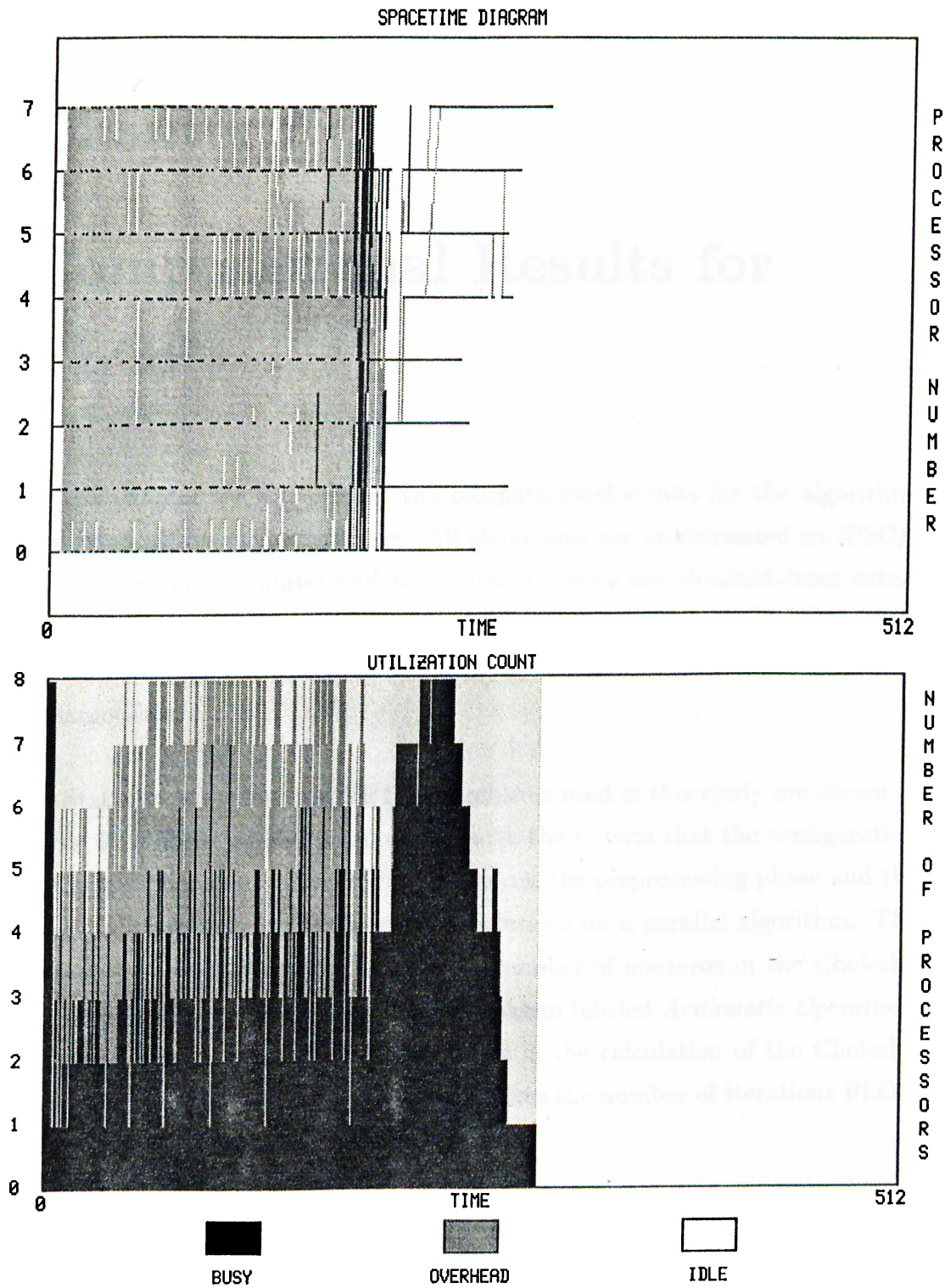


Figure 4.21: Spacetime diagram and utilization count for BFBS on 80bau3b problem.

Chapter 5

Computational Results for PLOP

In this chapter we will present the computational results for the algorithms presented in the earlier chapters. All algorithms are implemented on iPSC/2 hypercube multicomputer and performance results are obtained from actual runs on linear programming problems from the NETLIB suite [8]. These data represent realistic problems in industry applications ranging from small-scale to large-scale.

Statistics regarding the NETLIB problems used in this study are shown in Table 5.1. These problems are chosen with the criteria that the configuration of the host machine we used allows us to run the preprocessing phase and the problem is not very small to derive conclusions on a parallel algorithm. The column labeled *Nonzeros* in **L** gives the number of nonzeros in the Cholesky factor **L** (excluding the diagonal). The column labeled *Arithmetic Operations* gives the number of arithmetic operations in the calculation of the Cholesky factor **L**. And the column with label *Itr* gives the number of iterations PLOP executes with the problem.

Tables 5.2 and 5.3 show the change in factorization times and triangular solution (FS and BS) times by balance ratio of the task scheduling algorithm, respectively. All times given in this chapter are in milliseconds and are obtained by iPSC's `mclock()` function. From this data we conclude that a balance ratio of 0.80 gives the highest performance, though this choice does not have a major

effect. In Table 5.4 factorization times of fan-in algorithm are given. The numbers below each of the given times are speed-up values. Speed-up values are calculated as the ratio of parallel time to the time on one processor. This table shows that, the efficiency of the fan-in factorization algorithm is dependent on the problem size. That is, as the number of floating point operations needed in the factorization operations increases, speed-up values obtained by the parallel algorithm increases, too.

In Tables 5.5 and 5.6 computational results for two forward solution algorithms are given. They show that fan-in FS algorithm has better performance, as the number of processors increases, over the EBFS algorithm. Tables 5.7 – 5.9 give the results for three backward solution algorithms. Among these, buffered BS algorithm has the highest performance most of the time.

Computational results for sparse matrix-vector product $\mathbf{A}\delta$ and sparse matrix-matrix multiplication $\mathbf{A}\mathbf{D}\mathbf{A}^T$ are given in Tables 5.10 and 5.11, respectively. Both of these operations have satisfactory performance results. Matrix-vector product operation requires global collect of a vector at the beginning. Results obtained for this operation shows that, as the size of this vector increases, obtained speed-up values decreases. For example the data in Table 5.10 is obtained from the multiplication $\mathbf{A}\mathbf{x}$, and we see that as n (size of \mathbf{x}) increases, the performance decreases. Since $\mathbf{A}\mathbf{D}\mathbf{A}^T$ operation can't be executed on small number of processors, because of insufficient memory, its speed-up and percent efficiency values are calculated with respect to the time obtained on the smallest number of processors. Figs. 5.1 through 5.8 present the speed-up curves for the algorithms discussed.

Finally, in Table 5.12 and Fig. 5.9 the overall performance of one iteration of PLOP is presented. As the conclusion of our earlier results we used fan-in factorization, fan-in FS (FIFS) and buffered BS (BFBS) algorithms in our final implementation of PLOP. We used 0.80 as the balance ratio tolerance in the task scheduling algorithm and requested 8 significant figures in the objective value. These results show that, though PLOP performs well on moderate size problems, as the problem size increases the performance of PLOP increases, too. The resemblance between the speed-up curves of Fig. 5.9 and Fig. 5.1 for individual problems shows us that, general performance of a parallel interior

point algorithm is heavily dependent on the performance of the parallel factorization operation.

So far, we have not considered the issue of stability. Vanderbei [27] note that the stability of interior point methods needs to be better understood. He tells that many of the NETLIB problems results in poorly conditioned matrices as the method progresses. PLOP stops prematurely on problems `80bau3b` and `ship121` because of the stability issues, though it converges to the optimum value with 7 significant figures on the problem `ship121`.

Table 5.1: Statistics for the NETLIB problems used in this work.

Problem Name	Constraints (m)	Variables (n)	Nonzeros in A	Nonzeros in L	Operation count	Itr
bnl2	2324	3489	13999	83008	13593064	40
pilotnov	975	2172	13057	54753	6196921	29
pilot.ja	940	1988	14698	53738	5950772	39
cycle	1903	2857	20720	73427	5434413	32
woodw	1098	8405	37474	47757	3280086	27
80bau3b	2262	9799	21002	42510	2832016	29
25fv47	821	1571	10400	33465	2544443	29
maros	846	1443	9614	25367	1547758	28
nesm	662	2923	13288	23478	1465390	37
pilot4	410	1000	5141	14601	1020002	39
sctap3	1480	2480	8874	17362	617740	16
ship12l	1151	5427	16170	11189	182499	24

Table 5.2: Change of factorization times and separator sizes by balance ratio.
(For 16 processors)

Problem Name	Balance Ratio (min/max)		
	0.70	0.80	0.90
bnl2	8951	8855	8890
	723	729	736
woodw	3452	3461	3415
	373	375	457
80bau3b	2658	2700	2627
	276	286	295
25fv47	2558	2490	2526
	324	330	332

Table 5.3: Change of (FS+BS) times by balance ratio. (For 16 processors)

Problem Name	Balance Ratio (min/max)		
	0.70	0.80	0.90
bnl2	798	780	793
woodw	566	559	586
80bau3b	487	498	488
25fv47	443	438	447

Table 5.4: Computation times and speed-up values for fan-in factorization.

Problem Name	Number of Processors				
	1	2	4	8	16
bnl2	103209	55658	29472	14568	8854
		1.85	3.50	7.08	11.66
pilotnov	46869	28617	16014	8359	4905
		1.64	2.93	5.61	9.56
pilot.ja	45051	28462	15049	8443	4772
		1.58	2.99	5.34	9.44
cycle	41425	24596	17068	10360	6195
		1.68	2.43	4.00	6.69
woodw	25048	13548	10022	5530	3455
		1.84	2.50	4.53	7.25
80bau3b	21768	13139	7682	4010	2694
		1.66	2.83	5.43	8.08
25fv47	19270	10749	7150	3978	2487
		1.79	2.69	4.84	7.75
maros	11756	6701	3630	2828	1828
		1.75	3.24	4.16	6.43
nesm	11102	8329	3990	2441	1769
		1.33	2.78	4.55	6.28
pilot4	7709	4692	2614	1622	1085
		1.64	2.95	4.75	7.11
sctap3	4844	2979	1640	1156	866
		1.63	2.95	4.19	5.59
ship12l	1545	1120	838	959	443
		1.38	1.84	1.61	3.49

Table 5.5: Computation times and speed-up values for fan-in FS (FIFS).

Problem Name	Number of Processors				
	1	2	4	8	16
bnl2	1717	1069	691	462	351
		1.60	2.48	3.72	4.89
pilotnov	1111	745	524	362	287
		1.49	2.12	3.07	3.87
pilot.ja	1090	759	492	360	277
		1.44	2.22	3.03	3.94
cycle	1500	992	822	602	454
		1.51	1.82	2.49	3.30
woodw	969	576	408	318	260
		1.68	2.38	3.05	3.73
80bau3b	913	681	383	263	198
		1.34	2.38	3.47	4.61
25fv47	684	451	333	259	216
		1.52	2.05	2.64	3.17
maros	528	377	257	216	184
		1.40	2.05	2.45	2.87
nesm	485	348	269	213	183
		1.39	1.80	2.28	2.65
pilot4	298	220	156	133	108
		1.35	1.91	2.24	2.76
sctap3	402	287	230	192	163
		1.40	1.75	2.09	2.47
ship121	265	184	132	228	161
		1.44	2.01	1.16	1.65

Table 5.6: Computation times and speed-up values for EBFS.

Problem Name	Number of Processors				
	1	2	4	8	16
bn12	1433	889	496	338	452
		1.61	2.89	4.24	3.17
pilotnov	911	562	347	273	369
		1.62	2.63	3.34	2.47
pilot.ja	895	580	323	272	354
		1.54	2.77	3.29	2.53
cycle	1263	829	433	304	373
		1.52	2.92	4.15	3.39
woodw	811	464	275	204	264
		1.75	2.95	3.98	3.07
80bau3b	797	588	291	220	285
		1.36	2.74	3.62	2.80
25fv47	570	350	202	177	253
		1.63	2.82	3.22	2.25
maros	443	278	163	170	244
		1.59	2.72	2.61	1.82
nesm	405	260	176	167	258
		1.56	2.30	2.43	1.57
pilot4	250	172	118	123	185
		1.45	2.12	2.03	1.35
sctap3	356	222	153	119	155
		1.60	2.33	2.99	2.30
ship121	239	133	82	107	192
		1.80	2.91	2.23	1.24

Table 5.7: Computation times and speed-up values for EBBS.

Problem Name	Number of Processors				
	1	2	4	8	16
bnl2	1687	1459	1303	1170	1304
		1.16	1.29	1.44	1.29
pilotnov	1061	851	824	745	858
		1.25	1.29	1.43	1.24
pilot.ja	1039	824	750	728	846
		1.26	1.39	1.43	1.23
cycle	1474	1055	673	630	711
		1.40	2.19	2.34	2.07
woodw	945	610	455	425	507
		1.55	2.08	2.22	1.86
80bau3b	948	722	506	499	581
		1.31	1.87	1.90	1.63
25fv47	663	518	428	405	479
		1.28	1.55	1.64	1.38
maros	523	419	345	357	446
		1.25	1.52	1.47	1.17
nesm	473	369	372	378	471
		1.28	1.27	1.25	1.00
pilot4	291	270	241	269	323
		1.08	1.21	1.08	0.90
sctap3	430	299	229	195	230
		1.44	1.88	2.21	1.87
ship12l	288	161	94	265	617
		1.79	3.06	1.09	0.47

Table 5.8: Computation times and speed-up values for send-forward BS (SFBS).

Problem Name	Number of Processors				
	1	2	4	8	16
bnl2	1694	998	639	540	783
		1.70	2.65	3.14	2.16
pilotnov	1097	726	463	470	660
		1.51	2.37	2.33	1.66
pilot.ja	1075	669	418	459	655
		1.61	2.57	2.34	1.64
cycle	1490	950	645	676	945
		1.57	2.31	2.20	1.58
woodw	962	540	350	401	556
		1.78	2.75	2.40	1.73
80bau3b	906	633	348	319	465
		1.43	2.60	2.84	1.95
25fv47	674	397	288	315	478
		1.70	2.34	2.14	1.41
maros	520	308	222	282	452
		1.69	2.34	1.84	1.15
nesm	478	328	236	316	486
		1.46	2.03	1.51	0.98
pilot4	295	193	163	200	336
		1.53	1.81	1.48	0.88
sctap3	387	238	197	194	266
		1.63	1.96	1.99	1.46
ship121	255	151	100	242	350
		1.69	2.55	1.05	0.73

Table 5.9: Computation times and speed-up values for buffered BS (BFBS).

Problem Name	Number of Processors				
	1	2	4	8	16
bnl2	1545	1020	631	440	363
		1.51	2.45	3.51	4.26
pilotnov	996	677	449	348	271
		1.47	2.22	2.86	3.68
pilot.ja	976	626	406	340	280
		1.56	2.40	2.87	3.49
cycle	1358	883	641	477	427
		1.54	2.12	2.85	3.18
woodw	878	521	331	283	258
		1.69	2.65	3.10	3.40
80bau3b	829	595	349	256	220
		1.39	2.38	3.24	3.77
25fv47	613	375	281	223	204
		1.63	2.18	2.75	3.00
maros	473	311	217	202	202
		1.52	2.18	2.34	2.34
nesm	435	340	234	195	186
		1.28	1.86	2.23	2.34
pilot4	268	184	147	129	121
		1.46	1.82	2.08	2.21
sctap3	354	228	190	147	129
		1.55	1.86	2.41	2.74
ship121	233	146	97	156	156
		1.60	2.40	1.50	1.50

Table 5.10: Computation times and speed-up values for the computation of matrix-vector product $\mathbf{A}\delta$.

Problem Name	Number of Processors				
	1	2	4	8	16
bnl2	317	166	90	53	45
		1.91	3.52	5.98	7.05
pilotnov	269	139	75	44	28
		1.94	3.59	6.11	9.61
pilot.ja	293	152	80	47	29
		1.93	3.66	6.23	10.10
cycle	419	216	119	63	48
		1.94	3.52	6.65	8.73
woodw	788	409	217	121	106
		1.93	3.63	6.51	7.43
80bau3b	552	292	162	128	100
		1.89	3.41	4.31	5.52
25fv47	211	110	59	34	27
		1.92	3.58	6.21	7.81
maros	195	101	55	35	24
		1.93	3.55	5.57	8.13
nesm	285	149	80	47	40
		1.91	3.56	6.06	7.13
pilot4	109	57	32	18	13
		1.91	3.41	6.06	8.38
sctap3	202	106	58	34	23
		1.91	3.48	5.94	8.78
ship121	376	198	108	79	61
		1.90	3.48	4.76	6.16

Table 5.11: Computation times (T,in milliseconds), speed-up (S) and percent efficiency (E) values for the computation of matrix-matrix product \mathbf{ADA}^T .

Problem Name		Number of Processors				
		1	2	4	8	16
bnl2	T		780	457	260	167
	S			1.71	3.00	4.97
	E			85	75	62
pilotnov	T		1173	791	407	276
	S			1.48	2.88	4.25
	E			74	72	53
pilot.ja	T		2686	1639	824	617
	S			1.64	3.26	4.35
	E			82	82	54
cycle	T			1108	666	338
	S				1.66	3.28
	E				83	82
woodw	T			839	475	298
	S				1.77	2.82
	E				88	70
80bau3b	T			444	289	199
	S				1.54	2.23
	E				77	56
25fv47	T	1177	677	342	212	144
	S		1.74	3.44	5.55	8.17
	E		87	86	69	51
maros	T	1161	607	342	206	137
	S		1.91	3.39	5.63	8.47
	E		96	85	70	53
nesm	T	1279	747	432	218	130
	S		1.71	2.96	5.87	9.84
	E		86	74	73	62
pilot4	T	764	445	226	134	100
	S		1.72	3.38	5.70	7.64
	E		86	85	71	48
sctap3	T	634	355	241	138	88
	S		1.79	2.63	4.59	7.20
	E		89	66	57	45
ship121	T	781	427	258	150	110
	S		1.83	3.03	5.21	7.10
	E		92	76	65	44

Table 5.12: Computation times (T,in milliseconds), speed-up (S) and percent efficiency (E) values for one iteration of PCIPA (PLOP).

Problem Name		Number of Processors				
		1	2	4	8	16
bnl2	T		61643	33121	17790	10024
	S			1.86	3.47	6.15
	E			93	87	77
pilotnov	T		33173	18662	10307	6061
	S			1.78	3.22	5.47
	E			89	80	68
pilot.ja	T		53670	28474	16494	9538
	S			1.89	3.25	5.63
	E			94	81	70
cycle	T			34182	22124	12976
	S				1.55	2.63
	E				77	66
woodw	T			14528	8392	5338
	S				1.73	2.72
	E				87	68
80bau3b	T			12023	6773	4398
	S				1.78	2.73
	E				89	68
25fv47	T	25512	13847	9047	5300	3448
	S		1.84	2.82	4.81	7.40
	E		92	70	60	46
maros	T	17148	9633	5271	4056	2730
	S		1.78	3.25	4.23	6.28
	E		89	81	53	39
nesm	T	18272	12212	6272	3939	2770
	S		1.50	2.91	4.64	6.60
	E		75	73	58	41
pilot4	T	11062	6639	3699	2444	1705
	S		1.67	2.99	4.53	6.49
	E		83	75	57	41
sctap3	T	10113	5837	3335	2350	1700
	S		1.73	3.03	4.30	5.95
	E		87	76	54	37
ship12l	T	8605	4882	2948	2643	1636
	S		1.76	2.92	3.26	5.26
	E		88	73	41	33

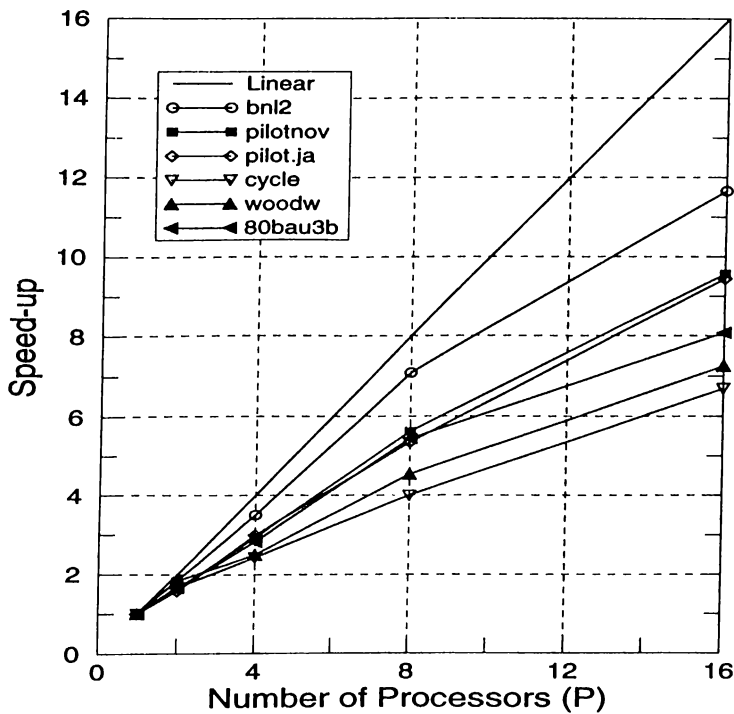


Figure 5.1: Speed-up curves for fan-in factorization algorithm.

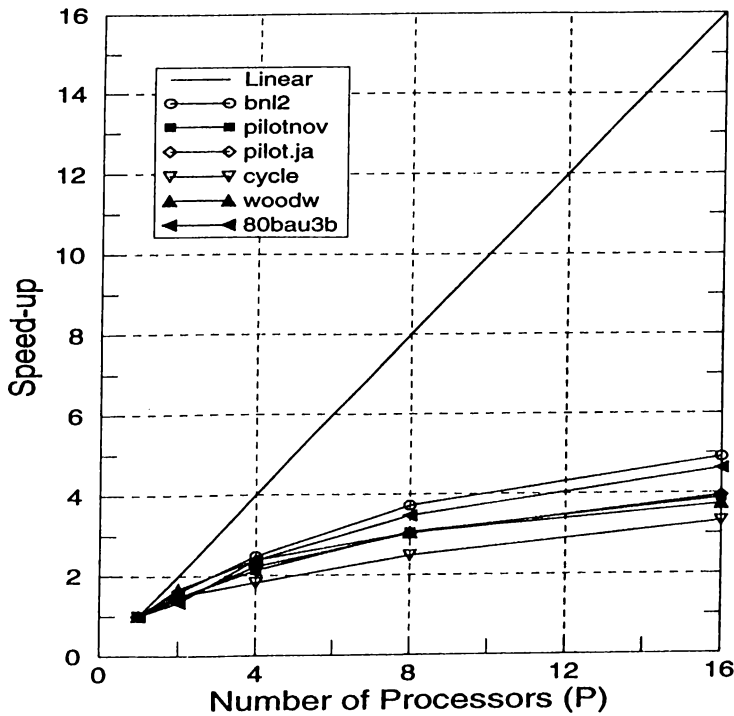


Figure 5.2: Speed-up curves for FIFS algorithm.

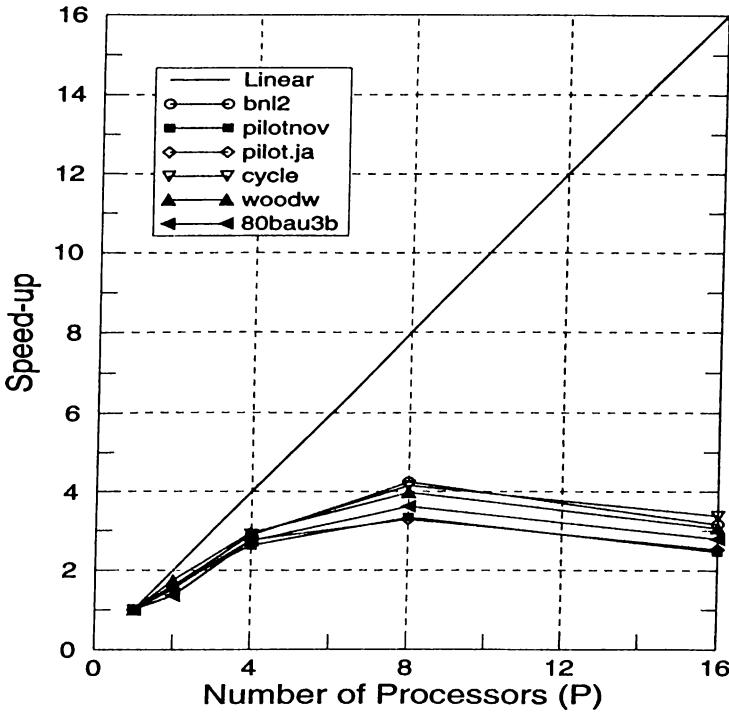


Figure 5.3: Speed-up curves for EBFS algorithm.

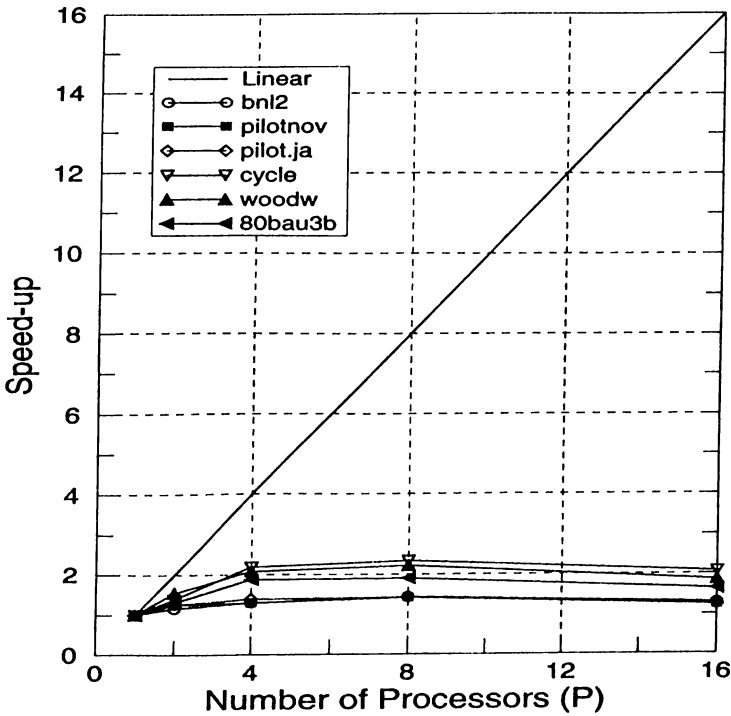


Figure 5.4: Speed-up curves for EBBS algorithm.

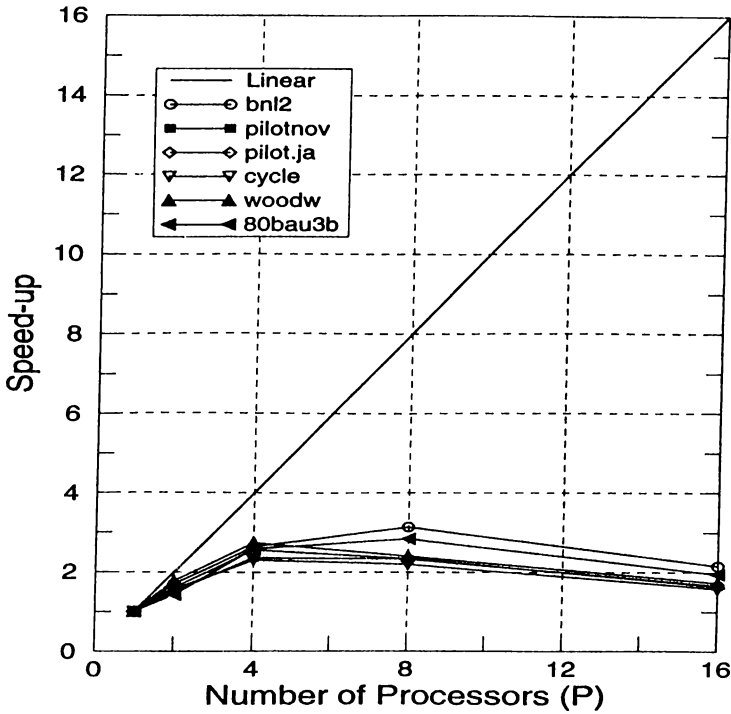


Figure 5.5: Speed-up curves for SFBS algorithm.

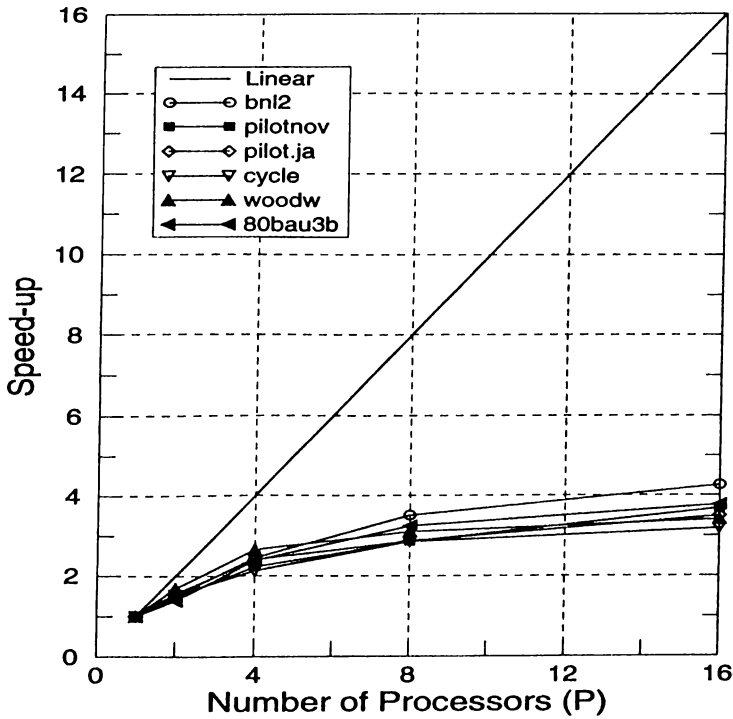
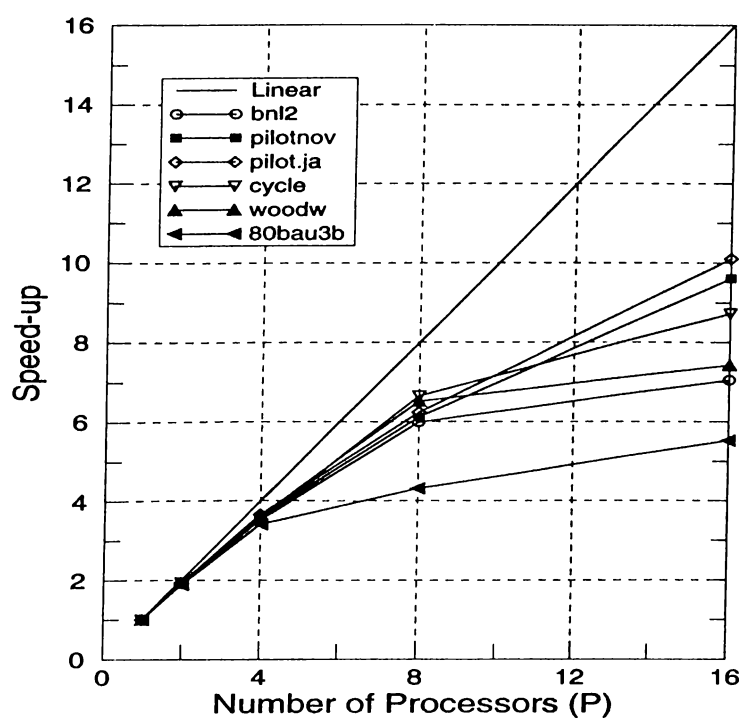
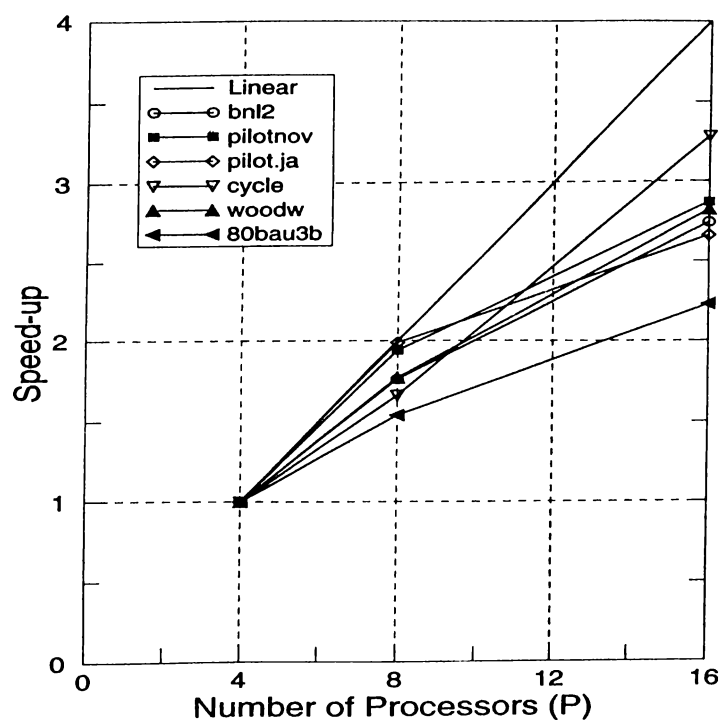


Figure 5.6: Speed-up curves for BFBS algorithm.

Figure 5.7: Speed-up curves for matrix-vector product $A\delta$.Figure 5.8: Speed-up curves for matrix-matrix product ADA^T .

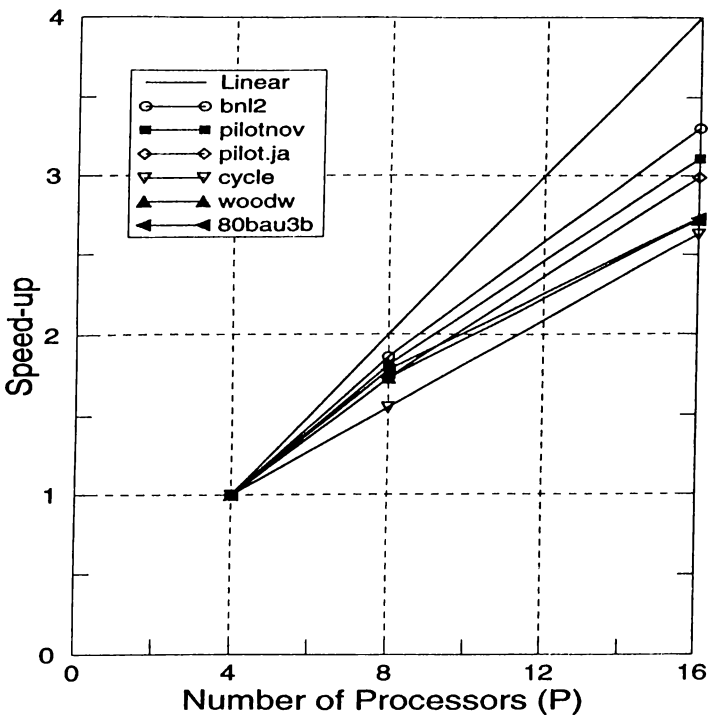


Figure 5.9: Speed-up curves for one iteration of PLOP.

Chapter 6

Conclusions

In this study, we presented the parallelization of a Karmarkar-type optimization algorithm — Mehrotra’s predictor-corrector interior point algorithm (PCIPA), on distributed memory message-passing parallel computers. Currently, literature doesn’t contain a parallel interior point algorithm for these architectures.

We identified five computation types needed by PCIPA as sparse matrix-vector product, vector operations, sparse matrix-matrix product, scalar operations, and sparse linear system solution. Then parallel algorithms for each type are presented. We have proposed a heuristic bin-packing algorithm, which is a row-level granularity scheduling method, to map rows and columns of a matrix for matrix-vector product operations. These mappings are used to distribute the vectors among the processors, too. Our experiments showed that this algorithm gives mappings which achieve load balance in the matrix and vector operations.

The solution of positive definite systems of linear equations constitutes the major computational effort in Karmarkar-type algorithms. In two major computation-intensive areas associated with the solution of such systems of linear equations, Cholesky factorization and forward and backward solvers, we have experimented with a number of algorithms to find the appropriate ones for interior point algorithms. For parallel sparse Cholesky factorization fan-in algorithm is used. This algorithm tries to reduce the amount of communication required during the factorization process. By this property it achieves better performance results than the fan-out algorithm, which has a high communication need. Another heuristic bin-packing algorithm is used to map columns of

this positive definite system for factorization. Here, elimination tree concept is used and subtrees are mapped to processors. Subtree mapping reduces the communication need of the factorization.

For forward solution we have compared elimination tree based forward solution and fan-in forward solution algorithms. For backward solution we implemented send-forward backward solution, elimination tree based backward solution, and buffered backward solution algorithms. Elimination tree based solution algorithms reduces the communication count, but they cause idle waitings. Though they gave better speed-up values for small number of processors, they didn't scale well when larger number of processors are used. Study of the spacetime diagrams of fan-in forward solution shows that this algorithm requires a huge amount of communication¹, but doesn't cause idle waitings as much as the elimination tree based forward solution. Hence, fan-in forward solution scaled well for larger number of processors. The situation is same between elimination tree based backward solution and buffered backward solution, and buffered backward solution gave better speed-up values as the number of processors increased.

All best performing algorithms are combined and a parallel linear optimizer (PLOP) program is implemented on iPSC/2 multicomputer. PLOP showed satisfactory overall performance on linear programming problems taken from NETLIB suite, which represent real industrial problems.

We believe that the algorithms and the methodology discussed in this work are general enough to be adapted for use with other versions of interior point algorithms and optimization methods. In summary, the preliminary implementation we have documented here shows sufficient promise of parallel execution of interior point algorithms, but as the previous chapter demonstrates, much further research will be required to produce a fully efficient implementation.

¹Actually, communication count of fan-in forward solution is equal to communication count of fan-in factorization.

References

- [1] Ilan Adler, Narendra Karmarkar, Mauricio G. C. Resende, and Geraldo Veiga. Data structures and programming techniques for the implementation of Karmarkar's algorithm. *ORSA Journal on Computing*, 1(2, Spring):84–106, 1989.
- [2] Ilan Adler, Narendra Karmarkar, Mauricio G. C. Resende, and Geraldo Veiga. An implementation of Karmarkar's algorithm for linear programming. *Mathematical Programming*, 44:297–335, 1989.
- [3] Cleve Ashcraft, Stanley C. Eisenstat, and Joseph W. H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM J. SCI. COMPUT.*, 11(3):593–599, May 1990.
- [4] Yun Chian Cheng, David J. Houck, et al. The AT&T KORBXC[©] system. *AT&T Technical Journal*, pages 7–19, May 1989.
- [5] In Chan Choi, Clyde L. Monma, and David F. Shanno. Further development of a primal–dual interior point method. *ORSA Journal on Computing*, 2:304–311, 1990.
- [6] Guy de Ghellinck and Jean-Philippe Vial. A polynomial Newton method for linear programming. *Algorithmica*, 1:425–453, 1986.
- [7] Jonathan Eckstein. Large-scale parallel computing, optimization, and operations research: A survey. *ORSA CSTS Newsletter*, 14(2):1,8–12,25–28, 1993.
- [8] D. M. Gay. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter*, 1985.
- [9] David M. Gay. Stopping tests that compute optimal solutions for interior-point linear programming algorithms. Num. Analy. Manuscript 89-11, AT&T, December 1989.

- [10] G. A. Geist and Esmond Ng. Task scheduling for parallel sparse cholesky factorization. *International J. of Parallel Programming*, 18(4):291–314, 1989.
- [11] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. Sci. Statist. Comput.*, 9:327–340, 1988.
- [12] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Solution of sparse positive definite systems on a hypercube. *J. of Comp. and Applied Math.*, 27:129–156, 1989.
- [13] Alan George and Joseph Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [14] Alan George and Joseph W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM REVIEW*, 31(1):1–19, March 1989.
- [15] S. Kingsley Gnanendran and James K. Ho. Load balancing in the parallel optimization of block-angular linear programs. *Mathematical Programming*, 62:41–62, 1993.
- [16] Michael Heath. Visual animation of parallel algorithms for matrix computations. In *Proc. Fifth Distributed Memory Computing Conf.*, 1990.
- [17] Michael T. Heath, Esmond Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM REVIEW*, 33(3):420–460, September 1991.
- [18] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno. On implementing Mehrotra’s predictor-corrector interior-point method for linear programming. *SIAM J. Optimization*, 2(3):435–449, August 1992.
- [19] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.
- [20] P. Sreenivasa Kumar, M. Kishore Kumar, and A. Basu. Parallel algorithms for sparse triangular system solution. *Parallel Computing*, 19:187–196, 1993.

- [21] Yong Li, Min You Wu, Wei Shu, and Geoffrey C. Fox. Linear programming algorithms and parallel implementations. Technical Report SCCS-288, Syracuse University, May 1992.
- [22] Roy Marsten, Radhika Subramanian, Matthew Saltzman, Irvin Lustig, and David Shanno. Interior point methods for linear programming: Just call Newton, Lagrange, and Fiacco and McCormick! *Interfaces*, 20(4):105–116, July 1990.
- [23] Kevin A. McShane, Clyde L. Monma, and David Shanno. An implementation of a primal-dual interior point method for linear programming. *ORSA Journal on Computing*, 1(2):70–83, 1989.
- [24] Sanjay Mehrotra. Implementations of affine scaling methods: Approximate solutions of systems of linear equations using preconditioned conjugate gradient methods. *ORSA Journal on Computing*, 4(2):103–118, 1992.
- [25] R. B. Panwar and P. Mazumder. A parallel Karmarkar algorithm on orthogonal tree networks. In *Int. Conf. on Parallel Proc.*, volume III, pages 274–276, 1990.
- [26] Robert J. Vanderbei. A brief description of ALPO. *OR Letters*, 10:531–534, December 1991.
- [27] Robert J. Vanderbei and Tamra J. Carpenter. Symmetric indefinite systems for interior point methods. *Mathematical Programming*, 58:1–32, 1993.
- [28] Youfeng Wu and Ted G. Lewis. Parallel algorithms for decomposable linear programs. In *Int. Conf. on Parallel Processing*, volume III, pages 27–34, 1990.