Sender: Dr. O. Knoth

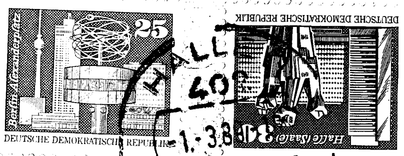
Martin-Luther-Universität
Halle-Wittenberg
Sektion Mathematik
Universitätsplatz 6
DDR - Halle
4 0 2 0
Deutsche Demokratische Republik

MIT LUFTPOST
PAR AVION

Mit Luftpost
Par avion

IV/10/23 P 792/86 4000 7300

Postkarte


Berlin Alexanderplatz
25
DEUTSCHE DEMOKRATISCHE REPUBLIK
DEUTSCHE DEMOKRATISCHE REPUBLIK
Halle (Saale)
30
HALL
402
1.-3.8

Dr. E.C.L. Chu
University of Waterloo
Department of Computer Science
Waterloo, Ontario N2L 3G1
Kanada - Canada

Halle, 27.2.89

Dear Dr. Chu

I should greatly appreciate receiving a reprint of your paper entitled:

Orthogonal decomposition of dense and sparse matrices on multiprocessors

which appeared in Ph. D. Thesis, R88/8

(and also of any other related papers).

With many thanks for your kindness,

Yours sincerely,

O. Knoth

88-08

sent April 5/89

Dr. A. Tamme,

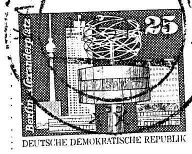
Absender: AdW der DDR,
Institut für Mechanik,
PF 408
Karl-Marx-Stadt

9 0 1 0

GDR

~~Postkarte~~

KARL ▼9010 ADT 1



Professor E.C.H. Chu

Department of Computer
Science,
University of Waterloo
Waterloo, Ontario
Canada - Kanada

Karl-Marx-Stadt, den 18.3.1989

Dear Professor Chu

Darf ich Sie höflichst um einen Sonderdruck Ihrer Arbeit bitten:
Je vous serais très obligé s'il vous était possible de m'adresser un tirage à part de votre publication:
I would be grateful if you could send me a reprint of your article entitled:
Разрешите попросить у Вас экземпляр Вашего труда:

"Orthogonal decomposition of dense and sparse Matrices on Multiprocessors"

and others

88-08

sent April 17/89

Mit bestem Dank im voraus und vorzüglicher Hochachtung
Je vous en remercie d'avance et vous prie d'agréer l'expression de mes sentiments très distingués
Thanking you in advance Yours sincerely
С благодарностью и уважением

[illegible]

VV Spremberg Ag 310 87 DDR 3092 I 20 8 721 Best.-Nr. 525 46

To Sue

From Eleanor Chu

Date Aug 24

memo

University of Waterloo

Please send report CS-88-08 "Orthogonal Decomposition of Dense and Sparse Matrices on Multiprocessors" in response to any request asking for report "Parallel algorithms for Linear Equations and least Squares problems", because we actually changed the title of the report after it was first referred to in one of our papers.

Thanks a lot.

Eleanor.

Printing Requisition/Graphic Services

15073

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION: CS-88-08

DATE REQUISITIONED: August 19/88 | DATE REQUIRED: ASAP | ACCOUNT NO.: 4 1 2 | 4 1 0 0 | 6 0

REQUISITIONER - PRINT: J.A. George | PHONE: | SIGNING AUTHORITY: [illegible]

MAILING INFO - NAME: Sue DeAngelis | DEPT.: C.S. | BLDG. & ROOM NO.: DC 2314 | [✓ DELIVER] PICK-UP

NUMBER OF PAGES: 198 | NUMBER OF COPIES: 50

TYPE OF PAPER STOCK: [✓ BOND] NCR ____ PT. [✓ COVER] BRISTOL [✓ SUPPLIED]

PAPER SIZE: [✓ 8½ x 11] 8½ x 14 11 x 17

PAPER COLOUR: [✓ WHITE] | INK: [✓ BLACK]

PRINTING: 1 SIDE ____ PGS. [✓ 2 SIDES] ____ PGS. | NUMBERING FROM TO

BINDING/FINISHING: [✓ COLLATING] STAPLING HOLE PUNCHED [✓ PLASTIC RING]

FOLDING/PADDING | CUTTING SIZE

Special Instructions

Math fronts and backs enclosed

Please bind

Thank you

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE: D01, D01, D01

TYPESETTING	QUANTITY	OPER. NO.	TIME	CODE
PAP000000				T01
PAP000000	[illegible]		[illegible]	T01
PAP000000				T01

PROOF				
PRF				
PRF				
PRF				

NEGATIVES	QUANTITY	OPER. NO.	TIME	LABOUR CODE
FLM				C01
FLM				C01
FLM				C01
FLM				C01
FLM				C01
PMT				
PMT				C01
PMT				C01
PMT				C01
PLATES				
PLT				P01
PLT				P01
PLT				P01
STOCK				
				001
				001
				001
				001
BINDERY				
RNG				B01
RNG				B01
RNG				B01
MIS000000				B01

OUTSIDE SERVICES

$ ____ COST

TAXES - PROVINCIAL ☐ FEDERAL ☐ GRAPHIC SERV. OCT. 85 482-2

UNIVERSITY OF ATHENS
PANEPISTEMIOPOLIS, 157 10 ATHENS
GREECE

DEPARTMENT OF MATHEMATICS
SECTION OF STATISTICS, O. R.
NUMERICAL ANALYSIS AND COMPUTER SCIENCE

TEL. (01) 72 43 219
72 43 246

Research Report Secretary
Department of Computer Science
University of Waterloo,
Waterloo.

May 18, 1988

88-08

Dear Madam,

I would like to order the following:

1) Parallel Algorithms for Linear Equations and Least Squares Proumblems by E. C. Chu, Ph.D. Thesis.

and

2) Fast Parallel Sorting on a Mesh-Connected Processor Array, by K. Sado and Y. Igarashi, CS-87-42

Could you please send me a (pro)-invoice with the total amount of payment for mailing there items to me? I will be much obliged.

Sincerely Yours
[illegible]

[illegible]

Orthogonal Decomposition Of Dense And Sparse Matrices On Multiprocessors

Eleanor Chin-hwa Lee Chu
Department of Computer Science

Research Report CS-88-08
March 1988

ORTHOGONAL DECOMPOSITION OF DENSE AND SPARSE MATRICES ON MULTIPROCESSORS

by

Eleanor Chin-hwa Lee Chu

A Thesis
presented to the University of Waterloo
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1988

ORTHOGONAL DECOMPOSITION OF DENSE AND SPARSE MATRICES ON MULTIPROCESSORS

Eleanor Chin-hwa Lee Chu
Department of Computer Science
University of Waterloo

Abstract

In this thesis we propose a number of new parallel algorithms for performing orthogonal decomposition of dense and sparse square or rectangular matrices. Our target machines are shared-memory multiprocessors and local-memory hypercube multiprocessors. For dense matrices, we propose an algorithm for shared-memory multiprocessors, and several algorithms for the hypercube multiprocessors. For sparse matrices, the algorithm we propose is specific to hypercubes. The paradigms we use in developing these parallel algorithms include divide-and-conquer, changing the order of computation, asynchronous computation and redundant computation. The algorithms designed for the hypercubes take further advantage of various topological properties of the network. We provide arithmetic and communication complexity analyses or implementations for each algorithm to indicate their expected performance. In particular, our analyses show that the parallel algorithms we propose for QR decomposition of dense (square or rectangular) matrices have *lower synchronization cost* or *lower communication cost* than other known schemes. These results are supported by numerical experiments.

Acknowledgements

I am deeply grateful to my teacher and thesis supervisor, Professor Alan George, for his guidance, encouragement and support throughout my graduate studies at the University of Waterloo. Professor George has inspired my interest in numerical linear algebra and parallel computing. His insight, enthusiasm and sound advice contributed invaluably to the completion and presentation of the work in this thesis.

Part of the research was conducted while I was visiting the Oak Ridge National Laboratory from October, 1986 to April, 1987. I wish to thank members of the Mathematical Sciences Section, especially Tom Dunigan, Young Etheridge, Al Geist, Michael Heath, Esmond Ng, Charles Romine and Robert Ward of the Computer Science Group. Their hospitality and friendship made my visit pleasant and productive, and their generosity in sharing with me their experience with the parallel computers is greatly appreciated. At a personal level, I thank Tom for providing me a hypercube simulator on the Sun workstation, which has facilitated my programming work greatly, and I thank Al, Chuck, Esmond and Mike for reading various parts of this thesis and offering very helpful comments and suggestions. I also wish to express my thanks to Professor Joseph Liu for reading an early version of this thesis and for his valuable advice and suggestions.

I thank Dr. Jack Dongarra and Professors Richard Bartels, Charles Colbourn, Bruce Simpson and Wei Pai Tang for serving on my committee.

Finally, thanks are due to my family for their love which had sustained me through all the years I was away pursuing my studies, and to all friends for their moral support.

The financial support of the University of Waterloo, the Natural Sciences and Engineering Research Council of Canada, and the Science Alliance, a state-supported program at the University of Tennessee is gratefully acknowledged.

Contents

1 Introduction **1**

1.1 The Mathematical Computation 1

1.2 Parallel Computers and Numerical Algorithms 3

1.3 Shared-Memory Multiprocessors 6

1.4 Local-Memory Multiprocessors 9

1.5 An Outline of The Thesis 12

2 QR Factorization of a Dense Matrix on a Shared-Memory Multiprocessor **15**

2.1 Introduction 15

2.2 The Algorithm 20

2.2.1 The Independent Annihilation Phase 20

2.2.2 The Cooperative Annihilation Phase 20

2.3 Implementation Issues 25

2.4 Analysis of Synchronization Cost 30

2.5 Analysis of Work Load Distribution 31

2.6 Performance Analysis 34

2.7 Numerical Experiments 37

2.8 Concluding Remarks 42

3 **QR Factorization of a Dense Matrix on a Hypercube Multiprocessor** **44**
3.1 Introduction 44
3.2 Algorithm I 46
3.3 Performance Analysis of Algorithm I 58
3.4 Algorithm II 60
3.5 Performance Analysis of Algorithm II 71
3.5.1 The case $m \geq n$ 72
3.5.2 The case $m \leq n$ 76
3.5.3 Analysis of Storage Requirements 82
3.6 Numerical Experiments 86
3.6.1 The Measurement of Serial Time 87
3.6.2 The Effect of the Aspect Ratio of the Processor Grid 89
3.6.3 Further Enhancement 95

4 **Sparse Orthogonal Decomposition on a Hypercube Multiprocessor** **105**
4.1 Introduction 105
4.2 Serial Row Merging Scheme 109
4.2.1 An Example 109
4.2.2 Definitions 117
4.2.3 The Algorithm 126
4.3 Parallel Row Merging Scheme 128
4.3.1 Basic Mapping Considerations 128
4.3.2 A Parallel Submatrix Merging Algorithm 131
4.3.3 Hypercube Partitioning 135
4.3.4 A Mapping Example 137
4.3.5 Complexity Analysis of a Model Problem 142
4.3.6 Generalizing The Parallel Submatrix Merging Algorithm. 165

5 Conclusions **169**

5.1 A Summary of Contributions 169

5.2 Further Work and Open Problems 171

Bibliography **177**

List of Tables

2.1 Execution time on the Balance 8000. 39

2.2 Execution time on the Balance 8000. 40

2.3 The effect of high synchronization cost. 41

3.1 Numerical solution to $f\left(p, \gamma_1, \frac{m}{n}, \frac{\tau}{\lambda}\right) = 0$. 76

3.2 Predicted optimal γ_1 when $m \geq n$. 77

3.3 Numerical solution to $\tilde{f}\left(p, \gamma_1, \frac{n}{m}, \frac{\tau}{\lambda}\right) = 0$. 80

3.4 Predicted optimal γ_1 when $m \leq n$. 81

3.5 Predicted and actual overhead storage. 85

3.6 Predicted γ_1^s and predicted optimal γ_1. 86

3.7 Execution times of the sequential Givens algorithm. 88

3.8 Measured and estimated times of the sequential Givens algorithm. 89

3.9 Single-precision execution times of Algorithm II. 91

3.10 Double-precision execution times of Algorithm II. 92

3.11 Estimated speed-up and efficiency of Algorithm II. 93

3.12 Estimated speed-up and efficiency of Algorithm II. 94

3.13 Execution times of the sequential Householder algorithm. 96

3.14 Measured and estimated times of the sequential Householder algorithm. . . 97

3.15 Single-precision execution times of the enhanced Algorithm II. 98

3.16 Double-precision execution times of the enhanced Algorithm II. 98
3.17 Estimated speed-up and efficiency of Algorithm II. 99
3.18 Estimated speed-up and efficiency of the enhanced Algorithm II. 100
3.19 Comparing the enhanced Algorithm II with other schemes. 103
3.20 Comparing the enhanced Algorithm II with other schemes. 104

List of Figures

1.1 Binary hypercubes of dimension 1, 2 and 3. 11

2.1 Standard Givens sequence [78]. 16
2.2 Dongarra et al. [13] and Lord et al. [64]. 17
2.3 Greedy Givens sequence [9,66]. 17
2.4 Independent annihilation by four processors. 21
2.5 CAP–initial data distribution among four processors. 23
2.6 CAP–after the annihilation of three main diagonals. 24
2.7 CAP–data mapping to eliminate the first superdiagonals. 26
2.8 End of IAP – a 16×8 example. 29
2.9 A multiprocessor model considered in [76]. 42

3.1 Column-by-column Givens sequence. 47
3.2 Wrap mapping of 16 rows to 4 processors. 49
3.3 The d communication steps in the CMP ($d = 3$). 50
3.4 The action of P_0 in the first elimination stage. 51
3.5 The distribution of local pivot rows at the end of Step 1 (IAP). 52
3.6 The 1^{st} elimination stage. 55
3.7 The 2^{nd} elimination stage. 55
3.8 The 3^{rd} elimination stage. 56

3.9 The embedding of a 4×4 grid in a hypercube. 61
3.10 The embedding of an 8×4 grid in a hypercube. 62
3.11 The wrap mapping of a 16×16 matrix to a 4×4 processor grid. 62
3.12 The wrap mapping of the submatrix within subcube $P(1,*)$. 63
3.13 Data assigned to processor P_0. 64
3.14 Data assigned to processor P_1. 64
3.15 Data assigned to processor P_2. 64
3.16 Data assigned to processor P_3. 65
3.17 The communication channels employed by the subcubes during the IAP. . 67
3.18 End of the IAP during the first elimination stage. 67
3.19 Data distribution in subcube $P(*,1)$. 68
3.20 Data to be processed by subcube $P(*,1)$. 68
3.21 Data distribution in subcube $P(*,2)$. 69
3.22 Data to be processed by subcube $P(*,2)$. 70
3.23 The communication channels employed by the subcubes during the CMP. . 70
3.24 Data distribution for the last four elimination stages. 72

4.1 Applying a Givens rotation to two sparse rows. 106
4.2 Sparse orthogonal reduction by Givens rotations. 106
4.3 A sequence of row merging operations. 107
4.4 A sparse Givens sequence. 107
4.5 A 16×9 example. 110
4.6 The George-Heath scheme. 111
4.7 The resulting upper triangular factor. 112
4.8 The row merging scheme - step 1. 113
4.9 The row merging scheme - step 2. 114
4.10 The row merging scheme - step 3. 115

4.11 The row merging scheme - the final step. 116
4.12 The first submatrix. 117
4.13 The second submatrix. 118
4.14 The third submatrix. 118
4.15 The fourth submatrix. 118
4.16 TZ[4,{3,6,7,8}]. 119
4.17 TZ[3,{5,7,8}], TZ[3,{5,8,9}], TZ[3,{6,7,8}] and TZ[3,{6,8,9}]. 120
4.18 The factor R from Figure 4.7 121
4.19 The elimination tree associated with the matrix A in Figure 4.5 121
4.20 The row merge tree associated with the matrix A in Figure 4.5 122
4.21 The binary row merge tree associated with the matrix A in Figure 4.5 . . . 123
4.22 The reduced row merge tree of A in Figure 4.5 124
4.23 Applying full matrix technique to submatrix merging. 125
4.24 A loop of four processors. 133
4.25 $X = TZ$[6,{1,2,3,4,5,6}]. 134
4.26 $Z = TZ$[6,{1,2,3,4,7,8}]. 134
4.27 Wrap mapping X and Z to a loop of four processors. 134
4.28 A snap shot. 135
4.29 The entire submatrix merging process. 136
4.30 Embedding loop(s) in a hypercube of dimension 4. 138
4.31 Nested dissection ordering of a 3-by-3 grid. 139
4.32 The 16×9 matrix associated with the grid in Figure 4.31 139
4.33 Nested dissection ordering of a 7-by-7 grid and separators. 140
4.34 An example of an elimination tree. 141
4.35 The (reduced) row merge tree associated with the 7-by-7 grid shown in Figure 4.33 145
4.36 A bordered subgrid from the 7-by-7 grid in Figure 4.33 146

4.37 Separators along the highest-cost path of a 15-by-15 grid. 147
4.38 An example of the highest-cost branch on a (reduced) row merge tree. . . . 147
4.39 Subproblems associated with subgrids. 149
4.40 Subproblems and column subscript sets for merging at vertex 37. 150
4.41 The four subgrids induced by the separators. 151
4.42 The submatrices $U = TZ[6,\{1,2,3,4,5,6\}]$ and $V = TZ[5,\{1,2,7,8,9\}]$. . . . 157
4.43 Merging $U = TZ[6,\{1,2, 3,4,5,6\}]$ and $V = TZ[5,\{1,2,7,8,9\}]$. 157
4.44 Merging $\tilde{U}$ and $\tilde{V}$. 158
4.45 Merging $\hat{U}$ and $\hat{V}$. 159

Chapter 1

Introduction

1.1 The Mathematical Computation

In this thesis we are concerned with designing parallel algorithms to perform orthogonal decomposition of rectangular matrices on multiprocessor computers. We first provide an overview of the mathematical computation we consider. It is well known that the orthogonal decomposition process is a basic tool for solving linear or nonlinear least squares problems and has applications to problems in linear programming, control, and eigenvalue and singular value computations. The orthogonalization of a rectangular matrix is usually formulated as

$$QA = \begin{pmatrix} R \\ 0 \end{pmatrix},$$

where A is an $m \times n$ $(m \geq n)$ matrix with full column rank, Q is an $m \times m$ orthogonal matrix and R is an upper triangular matrix of order n. The three principal methods for computing the factor R are Gram-Schmidt orthogonalization [2,75], Householder reflections [5,44,56], and Givens rotations [23,24,25,42,47]. The traditional implementations of both Gram-Schmidt and Householder compute R by eliminating the nonzero subdiagonal elements in an entire column of A at each step, whereas Givens rotations create one zero at a time by accessing a pair of rows. The element being annihilated by Givens rotations can be at any desired position of either row. These three methods are also different in their respective operation counts [43].

When A is dense, the choice of orthogonalization methods for implementation on a sequential computer is usually led by the following considerations. First, it depends on whether all the entries of A are available and if so whether the entire matrix can be stored in main memory. Second, we need to know whether the data are generated progressively during the orthogonalization phase and in what manner they are added to A. Third, the method's numerical stability, CPU cost and simplicity of coding must also be taken into consideration.

When A is sparse, regardless of which method is used, some zero entries in A become nonzero during the factorization process. If these entries are subsequently annihilated, they are referred to as *intermediate fills*. If they remain nonzero in the final structure of R, they are referred to as *fills*. Since the column ordering of A determines the sparsity structure of R, the fill in R is essentially independent of the orthogonalization method [28,45]. However, for any given A, assuming that the columns of A have been appropriately ordered to give a sparse R, the number of intermediate fills not only varies with the method but also may vary with different implementations of the same method. Therefore, when A is sparse there is added concern about the storage requirement and arithmetic operations caused by the intermediate fill.

The sparse orthogonalization method developed by George and Heath [28] applies Givens rotations to sparse rows in such a manner that the intermediate fills are always confined to a single working vector of length n and they suggest a suboptimal row ordering to limit the number of intermediate fills. Recently Liu proposed a general row merging scheme using Givens rotations [61]. The general row merging scheme processes the rows of A in a particular sequence. Experimental results provided in [61] indicate that substantial reduction in intermediate fills can be achieved compared to employing other heuristic row orderings. The tradeoff for the gain in execution time is a moderate increase in working storage [61].

A comprehensive survey of sparse orthogonalization methods developed between 1976 to 1983 is contained in [50]. Further progress made on this subject since 1983 can be found in [32,34,35,36,37,39,40,41]. Several interesting questions relating to sparse orthogonal factorization remain open today and this is an area of active current research.

1.2 Parallel Computers and Numerical Algorithms

When the dense or sparse coefficient matrix is large, the orthogonalization process is very demanding in CPU time and memory space on a sequential machine. For example, a system of overdetermined nonlinear equations arising from the geodetic least squares adjustment problem consists of over 6,500,000 equations in over 540,000 unknowns [46]. The general solution procedure using the Gauss-Newton method involves solving a sequence of linear least squares problems of the same size [45,46]. Very large sparse linear least squares problems also arise in earthquake studies, photogrammetry and certain types of finite element analysis. With the performance of a sequential computer limited by the circuit speed, the accurate solution of such problems in realistic time limits can only be realized by the effective use of parallel computers.

The ideal parallel algorithm is expected to solve a problem on a parallel computer having p processors p times faster than solving the same problem on a sequential machine having one processor of similar processing power. To achieve this goal on different types of parallel architecture requires different techniques which take into consideration the overall organization of the target machine. The main features of machine architecture that an algorithm designer should be aware of include: the number and type of processors, memory modules and input/output channels, and how these are controlled and interconnected. Along with the development and change of parallel architectures, the design of parallel numerical algorithms has been an active research area in the past two decades. However, since the VLSI (very large-scale integration) semiconductor technology for implementing the parallel architecture was not available until the mid-1970s, certain practical limits of the machine were unfortunately ignored in some early designs and complexity analyses for parallel numerical algorithms [26]. The survey articles by Heller [54] and Sameh [79] review parallel techniques for problems in numerical linear algebra developed before 1978. Two more recent surveys of parallel algorithms in this area are contained in [19] and [77]. A bibliography on parallel and vector numerical algorithms prepared by Ortega and Voigt [67] was published in 1987.

Since a wide variety of different parallel architectures have been proposed, it is useful to identify some characteristics which can classify them. Flynn's taxonomy is such a classification scheme based on how the machine relates its instructions to the data being processed [20]. He defines *a stream* as a sequence of items (instructions or data) as executed or operated on by a processor. Depending on the instructions or data streams being single or multiple, the designs of parallel architectures fall into two broad categories, for which we quote the definitions given in [55] below.

SIMD – *single instruction stream/multiple data stream. This is a computer that retains a single stream of instructions but has vector instructions that initiate many operations. Each element of the vector is regarded as a member of a separate data stream hence, excepting the degenerate case of vectors of length one, there are multiple data streams. This classification therefore includes all machines with vector instructions. It is irrelevant whether the capability of vector processing is realized by pipelining or by building arrays of processors.*

MIMD – *multiple instruction stream/multiple data stream. Multiple instruction streams imply the existence of several instruction processing units and therefore necessarily several data streams. This class therefore includes all* *forms of multiprocessor configurations, from linked main-frame computers to large arrays of microprocessors.*

By 1981 several SIMD computers had been offered on a commercial basis. Examples of pipelined computers are the CRAY-1, CDC CYBER 205 and FPS AP-120B, and examples of processor arrays are the ICL DAP [55,69] and versions of special-purpose systolic arrays [3,4,86]. By then it had also been noticed that the ratio between the performance of a good and a bad parallel implementation of the same serial algorithm could easily exceed a factor of ten or more on a real machine [55]. Instead of waiting for the "ideal" machine to be built, programming professionals, algorithm researchers and practising scientists have since devoted enormous effort to develop new parallel algorithms, parallel languages and software in order to take advantage of the advance in computer technology. At the same time, matrix computations continue to be one substantial application area for comparing the performance of the different designs. The principles of pipelined computers and processor arrays along with the algorithmic and language aspects of parallelism are well introduced in the book by Hockney and Jesshope [55]. The review article by Dongarra et al. [12] examines the common implementations of linear algebra algorithms for dense matrices on vector pipeline computers. The linear algebra algorithms designed for systolic arrays are surveyed by Schreiber in [81].

By 1985, several general-purpose multiprocessors became commercially available. They are all MIMD computers using Flynn's taxonomy. A finer classification of these machines based on memory organization distinguishes two major architectures: *shared-memory* multiprocessors, and *local-memory* (or *distributed-memory*) multiprocessors. Since we are concerned with designing orthogonal factorization algorithms for both shared-memory and local-memory multiprocessors, in the following sections we shall introduce each type of architecture and the paradigms for programming them.

1.3 Shared-Memory Multiprocessors

A shared-memory multiprocessor consists of multiple identical processors (CPUs) and a single common memory. Each CPU is a general-purpose computer which can execute both user code and operating system code. The shared-memory is typically very large and may be configured as a number of independent and interleaved memory modules. The full connection between the processors and all of the memory modules is realized by a switching network or a high-speed system bus. The contention for memory cycles by multiple processors is a concern in a shared-memory environment. In order to reduce the number of times an individual processor must use the bus to access the common memory, some systems equip each processor with both local RAM and cache RAM. They appear to be effective in supporting the bus architecture when the number of processors is not very large. The currently successful multiprocessors of this type typically have eight to thirty processors.

The number of processors used for an application is usually specified by the user. Each processor has an identity number which can be used by programmers to distinguish different actions for individual processors in the code. Therefore, even if the same program is run on all participating processors, the code actually executed by each processor could be different at times. Since each processor runs its own operating system and supports multiple processes, it is common to distinguish processors and processes in describing algorithms. However, since we are interested in designing parallel algorithms with all processes being executed simultaneously, only one process will be assigned to each available processor and the communication between any two processes are always controlled by two different processors. There is no chance of confusion between the two here. Therefore, in order to make explicit the actions performed by different processors, we shall always refer to the controlling processors in our discussion throughout this thesis.

Because all of the data in the common store are accessible to all processors working on the same problem, some accessing discipline must be exercised to maintain the data integrity. To ensure that the parallel algorithm does produce the correct answer, we are concerned with two aspects of data integrity. First, some data will be modified by multiple processors as the computation proceeds. Access to such data must only be granted to one processor at a time. In other words, the code which updates such shared data should only be executed by one processor at a time. Such code segments are referred to as "critical sections" in a parallel program. Secondly, the nature of the solution procedure usually requires the data to be modified in a particular order. The processors cooperating to solve the problem must synchronize with each other to satisfy such precedence constraints inherent in the algorithm.

To enable parallel programs to ensure that no two processors write to the same memory location at the same time, current systems commonly provide hardware support for the "lock" and "unlock" operations. If we associate with each shared data structure a *lock variable*, then a "lock" operation preceding the critical section guarantees that when the access is granted the processor executing the code in the critical section is the sole processor which is operating on the data. During this period any other processor executing the "lock" operation on the same lock variable will simply spin or busy wait. An "unlock" operation following the critical section resets the lock variable so that another processor may gain access to the protected data.

When multiple processors cooperate to solve a problem, it is common that the data one processor needs to work on at a later stage are produced by a different processor or a number of different processors at an earlier stage. Therefore, the processors must coordinate their actions so that the updates to the data are carried out in a correct sequence. Depending on whether the processors synchronize with each other *explicitly* or *implicitly*, we have either a *synchronous* algorithm or an *asynchronous* algorithm. In the former case, the parallel algorithm is usually expressed as a number of sequential stages which dictate the order the data are modified or produced, and the parallelism is obtained by having all processors work on disjoint subsets of data concurrently within each stage. Therefore, if the processors explicitly synchronize with each other by waiting at the end of each stage until the slowest processor completes its work for the current stage, it is guaranteed that the data is ready for the next stage of computation to proceed. A device that is commonly provided in the system library for explicit synchronization among multiple processors is the *barrier* function. The parameter to the *barrier* function is a counter variable which is initialized by the user to be the number of processors participating in the explicit synchronization. All processors stop at the barrier by calling the *barrier* function with the specified counter variable. The processors checking in earlier are suspended until the last one checks in. At that time all processors are released simultaneously to resume the computation.

In the case of implicit synchronization, the readiness of each data structure for a particular stage of computation is indicated by some global variables in the shared-memory. The status of such global variables are normally updated by the processor which is currently operating on the data structure to reflect its readiness for the future stages of computation. Therefore, there is much freedom in arranging the computations in an asynchronous algorithm to balance the work load and exploit the parallelism.

An integral part of our research is the complexity analysis of the synchronization cost, work load distribution and performance of the proposed algorithms. The following assumptions are implicitly made about the multiprocessor system in our analytical model.

1. All processors are running at the same speed.

2. The application program is the sole process running on each processor.

3. All processors participating in an application are operational throughout the entire computation.

4. The arithmetic cost is measured by the number of multiplicative floating-point operations.

5. The synchronization cost in addition to the arithmetic cost is analyzed separately.

1.4 Local-Memory Multiprocessors

A local-memory multiprocessor differs from a shared-memory multiprocessor in its distribution of memory to individual processors. Each processor on a local-memory machine executes its own program and operates on its own data out of its private memory. There is neither globally shared memory nor connection between one processor to another processor's memory modules. Instead, the processors are connected by a network of communication channels and they synchronize with each other by sending and receiving messages over the network. Depending on the circumstances, the message can either be data for another processor to work on or status information for every one on the network. This presents a different set of challenges for both the hardware designer and the algorithm designer.

For the hardware architect, tradeoff must be made between the cost of equipment and the richness of the connection topology. Networks of various types and topologies are used. They can be of fixed topology, such as a ring or mesh, or can be packet-switched or circuit-switched networks. The following examples are given in [21]. The Finite Element Machine [58] is a mesh-connected lattice of 36 processors. The Cal-Tech Cosmic Cube contains 64 processors connected as a binary hypercube of dimension 6. The Non-Von (Columbia University) is a tree of processors. The CHiP architecture is a lattice of processors [82] interconnected via a circuit-switching network which can be configured as any member of a large family of graphs. The Boolean Vector Machine (Duke University) is an implementation of the Cube Connected Cycle network [73]. The hypercube is probably the most common topology currently in use, and multiprocessors based on it are available from several commercial vendors.

The challenge to the algorithm designers is to partition the computation and data in such a way that each processor has the data it needs in its local memory at the time that it needs it, and that the time consumed by data transmission contributes little to the total time of the parallel algorithm. Since many networks allow the embedding of a variety of topologies, taking on this challenge on them on one hand permits more freedom in exploiting the parallelism in the numerical computation and on the other hand, the algorithm designers are faced with the task of envisioning a communication topology which is most efficient for the application. A hypercube network is an example in which various topologies (loop, mesh, tree, toroid, etc.) can be embedded. Since we are primarily interested in designing parallel algorithms for the hypercube, we shall briefly review the properties of the hypercube network in the following discussion.

A hypercube of dimension d consists of 2^d identical processors. Each processor is assigned a unique identifier (ID) from the integers in $\{0, 1, 2, \cdots, 2^d - 1\}$, which can each be represented as a d-bit binary string. A direct communication channel is provided for every pair of processors whose ID's binary representations differ in one single bit. Thus, each processor on a d-dimensional hypercube has exactly d neighbours, and a $(d+1)$-dimensional hypercube is constructed by the pairwise connection between the processors of two d-dimensional hypercubes. We depict in Figure 1.1 hypercubes of dimension 1, 2 and 3. The processor ID is used to specify the receiver of the message and distinguish code segments for different processors. If the specified receiver is not a neighbour of the sender, the message will be forwarded by the intermediate nodes. Letting p denote the total number of node processors on a d-dimensional hypercube, the longest communication path between two arbitrary nodes is d or $\log_2 p$.

Operating systems for local-memory multiprocessors normally provide message-passing primitives for sending and receiving data and synchronization information over the network. The two primitives which are most commonly used in coordinating the asynchronous actions by different processors are the non-blocking *send* and the blocking *receive*. Execution of a *send* does not cause the sending processor to wait for a reply. On the other hand, execution of a *receive* causes the processor executing it to be suspended until the message is received. Messages that arrive at the destination processor before the execution of the *receive* are placed in a queue until needed. They are the only message passing primitives we assume in

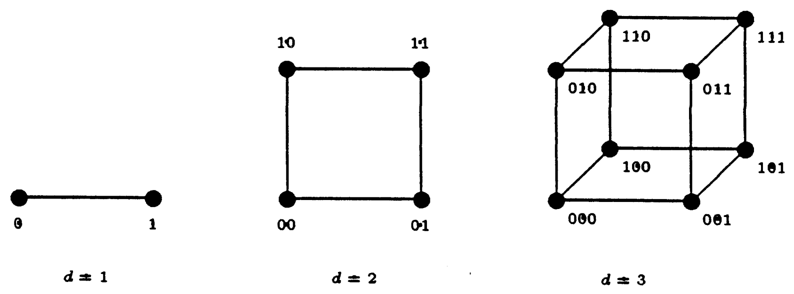


Figure 1.1: Binary hypercubes of dimension 1, 2 and 3.

developing the parallel algorithms for hypercubes in this thesis.

To analyze the communication cost and arithmetic cost of the algorithms we propose and implement on the hypercube multiprocessor, we model the system by the following rules.

1. The $p = 2^d$ processors of a d-dimensional hypercube are operational throughout the entire computation.

2. All processors are running at the same speed.

3. Each multiplicative floating-point operation takes τ units of time.

4. When processor P_i sends a message of n floating-point numbers to a directly connected neighbour P_j, the message is received by P_j after

$$\beta + n\lambda$$

units of time, where β is the start-up time and λ is the time for sending one floating-point number across one link in the network.

5. When two neighbouring processors exchange messages of size n_i and n_j, the time for the exchange is modeled as sending two messages sequentially across one link, namely

$$2\beta + (n_i + n_j)\lambda$$

units of time.

6. The ratio τ/λ is a hardware-dependent parameter in our model.

7. The time for data transmission is not overlapped by computation.

Since the commercial availability of hypercubes in 1985, there has been a conference devoted specifically to its architectures, programming environments and applications each year. The proceedings entitled "Hypercube Multiprocessors 1986" [48] and "Hypercube Multiprocessors 1987" [49] contain a hundred papers given at the first two conferences. The wide range of topics covered include architectures, operating systems, programming languages, algorithms, data structures, and applications. Special emphasis was given to matrix computations and partial differential equations in [49]. With the continued advancement in technology, the future generations of hypercubes are expected to have more powerful processors (and many more of them), larger memories, faster internode communication and external I/O capacity within each individual processor. Developing efficient algorithms and programming methodology to realize the supercomputer potential of this continuously evolving architecture is currently at the forefront of parallel processing research.

1.5 An Outline of The Thesis

In Chapter 2 a new algorithm for computing the orthogonal decomposition of a rectangular $m \times n$ matrix A on a shared-memory multiprocessor is described. The algorithm uses Givens rotations, and has the feature that its synchronization cost is low. In particular, our analysis indicates that the upper bound of the synchronization cost is *independent* of m, the row dimension of the matrix A. This is important for machines where synchronization cost is high, and when $m \gg n$. The work load distribution and the expected performance of the proposed algorithm are both analyzed. The algorithm is implemented in FORTRAN and tested on a Sequent Balance 8000 parallel computer, which is a bus-connected multiprocessor having 8 processors and 8M bytes of global memory. Timing results are provided for comparing synchronous and asynchronous implementations of the proposed algorithm, and our implementation of the pipelined Givens method proposed in [13].

In Chapter 3 we describe a new algorithm for computing the QR factorization of a rectangular matrix on a hypercube multiprocessor. The scheme involves the embedding of a two-dimensional grid in the hypercube network, where each row or column of the grid is an embedded subcube. We employ a global communication scheme which uses redundant computation to maintain data proximity, and the mapping strategy is such that the processor idle time remains constant (for square matrices) or small (for rectangular matrices) when the number of processors is fixed regardless of the size of the matrix. A complexity analysis tells us what the aspect ratio of the embedded grid should be in terms of the shape of the matrix and the relative speeds of communication and computation. Numerical experiments performed on an Intel Hypercube multiprocessor support the theoretical results.

In Chapter 4 we consider the QR decomposition of a class of large sparse matrices on a hypercube multiprocessor. The proposed scheme is essentially a parallel implementation of the general row merging scheme developed by Liu in [61] for sparse Givens transformations. We show how multiple loops of different sizes can be embedded in the hypercube network and how this novel topology is employed in the proposed algorithm. The performance of the proposed algorithm applied to a model problem is analyzed and computation/communication complexity results are presented.

The analyses given in Chapters 2, 3 and 4 were aided significantly by the use of MAPLE, an algebraic manipulation system developed at the University of Waterloo [7].

Chapter 5 contains our concluding remarks and a discussion of some related future research problems.

Chapter 2

QR Factorization of a Dense Matrix on a Shared-Memory Multiprocessor

2.1 Introduction

In this chapter we present an algorithm for reducing an $m \times n$ $(m \geq n)$ matrix to upper triangular form on a shared-memory multiprocessor having p identical processors. The use of Givens rotations has been studied for parallel implementation on shared-memory multiprocessors by other researchers in [9,13,64,66,76,78]. Since the parallel algorithm we describe is also based on Givens rotations, a brief review of their schemes will provide useful background information.

It is well-known that there is much freedom in the order of applying the Givens rotations. For a particular Givens ordering, the *theoretical* minimum number of parallel steps is obtained by assuming that all *independent* (or *disjoint*) rotations can be computed simultaneously in one step. The parallel algorithms presented in [9,13,64,66,76,78] are all based on "Givens sequences", that is, sequences of Givens rotations in which zeros once created are preserved. The question of whether temporarily annihilating elements and introducing zeros that are destroyed later can lead to any additional parallelism is discussed in [9]. The odd-even ordering used in the rotation method proposed by Luk in [65] has this property of creating redundant zeros.

Figures 2.1–2.3 illustrate the different Givens sequences used in [9,13,64,66,78] for an 8×8 matrix. For each sequence illustrated, the disjoint rotations are identified by the same step number. Since the elimination order is from left to right for all three Givens sequences, the first k ($k < 8$) columns of each matrix illustrate the annihilation ordering for an $8 \times k$ rectangular matrix.

$$\begin{pmatrix}
\times & \times & \times & \times & \times & \times & \times & \times \\
7 & \times & \times & \times & \times & \times & \times & \times \\
6 & 8 & \times & \times & \times & \times & \times & \times \\
5 & 7 & 9 & \times & \times & \times & \times & \times \\
4 & 6 & 8 & 10 & \times & \times & \times & \times \\
3 & 5 & 7 & 9 & 11 & \times & \times & \times \\
2 & 4 & 6 & 8 & 10 & 12 & \times & \times \\
1 & 3 & 5 & 7 & 9 & 11 & 13 & \times
\end{pmatrix}$$

Figure 2.1: Standard Givens sequence [78].

For the Givens sequences illustrated in Figures 2.1–2.3, $\lfloor m/2 \rfloor$ processors are required to factor an $m \times n$ matrix using the minimum parallel steps. The availability of $\lfloor m/2 \rfloor$ processors is assumed in the algorithm analyses in [9,66,78], and the optimality of the greedy Givens sequence is established in [9] under the same condition. The parallel algorithm proposed by Sameh and Kuck [78] is based on the standard Givens sequence (Figure 2.1), for which $(m + n - 2)$ steps are required to factor an $m \times n$ matrix using up to $\lfloor m/2 \rfloor$

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times & \times & \times \\ 1 & \times & \times & \times & \times & \times & \times & \times \\ 2 & 3 & \times & \times & \times & \times & \times & \times \\ 3 & 4 & 5 & \times & \times & \times & \times & \times \\ 4 & 5 & 6 & 7 & \times & \times & \times & \times \\ 5 & 6 & 7 & 8 & 9 & \times & \times & \times \\ 6 & 7 & 8 & 9 & 10 & 11 & \times & \times \\ 7 & 8 & 9 & 10 & 11 & 12 & 13 & \times \end{pmatrix}$$

Figure 2.2: Dongarra et al. [13] and Lord et al. [64].

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times & \times & \times \\ 3 & \times & \times & \times & \times & \times & \times & \times \\ 2 & 5 & \times & \times & \times & \times & \times & \times \\ 2 & 4 & 7 & \times & \times & \times & \times & \times \\ 1 & 3 & 6 & 8 & \times & \times & \times & \times \\ 1 & 3 & 5 & 7 & 9 & \times & \times & \times \\ 1 & 2 & 4 & 6 & 8 & 10 & \times & \times \\ 1 & 2 & 3 & 5 & 7 & 9 & 11 & \times \end{pmatrix}$$

Figure 2.3: Greedy Givens sequence [9,66].

processors. A parallel algorithm based on the greedy Givens sequence (Figure 2.3) was proposed independently by Modi et al. [66] and Cosnard et al. [10]. While there is no exact analysis, by assuming m goes to infinity with n fixed, Modi and Clarke's approximate analysis in [66] gives the asymptotic complexity of

$$\log_2 m + (n-1)\log_2 \log_2 m$$

parallel steps. Although the reduction in the number of steps from $(m+n-2)$ to $(\log_2 m + (n-1)\log_2 \log_2 m)$ is impressive, the efficiency of this algorithm is not satisfactory since $\lfloor m/2 \rfloor$ processors are used. For n *fixed*, $m \to \infty$, Cosnard et al. [9] derive an efficiency of

$$2n/\log_2 m + o(1/\log_2 m).$$

They assume that all rotations in the serial Givens scheme or in the parallel scheme take the same amount of time.

Cosnard et al. also derive the asymptotic complexity of $2n$ parallel steps for the case m/n^2 tending to zero as m and n go to infinity. In view of the relatively small number of processors on a shared-memory multiprocessor, it is unlikely that the assumption of $\lfloor m/2 \rfloor$ available processors will hold for any problem of reasonably large size. Therefore, these complexity results are mainly of theoretical interest.

The parallel algorithms proposed in [13] and [64] are designed for a shared-memory multiprocessor with low synchronization overhead. The parallel ZIGZAG scheme proposed by Lord et al. in [64] can be viewed as implementing the Givens sequence shown in Figure 2.2 on a square matrix of order n using $\lfloor n/2 \rfloor$ processors. The asymptotic efficiency of this algorithm is 44.4%. For $n = 17$, the actual efficiency of 45% approaches the asymptotic value [64]. Thus, the predicted (and actual) efficiency of this algorithm is low, although this is not surprising given the large number of processors employed. The COLSWP (column-sweep) Givens scheme proposed in [64] and the pipelined Givens method proposed in [13] assume a relatively small number of processors; i.e. , the number of processors is assumed to be much less than n. When p, the number of processors, is much less than $\lfloor m/2 \rfloor$, the parallelism allowed by a particular Givens sequence can be exploited in a variety of ways. Indeed the COLSWP algorithm can be viewed as implementing the Givens sequence shown in Figure 2.2 in a column-by-column manner so that the zero elements in each column are created by the same processor. For the pipelined Givens method, the same Givens sequence is implemented in a row-by-row manner so that the zero elements in each row are created by the same processor.

As discussed in Chapter 1, on a shared-memory computer with multiple processes running in parallel, the processes must be synchronized in order to prevent their simultaneously updating shared data and thereby corrupting it. This synchronization can be achieved through the use of "locks". A lock ensures that only one process at a time can access a shared data structure. A lock has two values: locked and unlocked. Before attempting to access a shared data structure, a process waits until the lock associated with the data structure is unlocked. The process then *locks* the lock, accesses the data structure, and *unlocks* the lock. In our analysis in section 2.4, we assume a fixed overhead for acquiring and releasing a lock and measure the synchronization cost by the number of times a lock is accessed. We do not include the time a processor may be waiting for the lock to be unlocked in the synchronization cost because such waiting time is accounted for in the performance analysis of the parallel algorithm given in section 2.6.

The synchronization cost of the parallel schemes in [13] and [64] is not analyzed. This is reasonable because *low* synchronization overhead was assumed. Our analysis of the pipelined Givens algorithm in [13] shows that its synchronization cost is a function of m, n, and p. The dependence of the synchronization cost on the row dimension m is undesirable when $m \gg n$, which is not uncommon. This prompted us to devise a parallel algorithm for which the synchronization cost is *independent* of the row dimension m. Such a scheme is particularly suitable for multiprocessors whose synchronization overhead is significant.

The algorithm we are going to propose in the next section is similar in several aspects to Sameh's 2-stage orthogonal factorization algorithm in [76]. After our algorithm is described and analyzed, we shall compare and contrast the two in the section containing our concluding remarks.

2.2 The Algorithm

The algorithm we propose has been designed for shared-memory multiprocessors. The main objective of our design is to reduce the synchronization cost and processor idle time by assigning the processors to work on disjoint sets of rows as much as possible. The algorithm has two phases: an independent annihilation phase (IAP) and a cooperative annihilation phase (CAP). For ease of exposition, we first describe the algorithm for factoring an $m \times n$ matrix A using p processors, where m and n are integral multiples of p, and $m/p \geq n$. For example, assuming $p = 4$, a matrix of dimension 32×8 satisfies the above condition.

2.2.1 The Independent Annihilation Phase

In the IAP, each processor is assigned a block of m/p consecutive rows of A. Each processor reduces its own block of rows to an upper triangular submatrix of order n ($n \leq m/p$ by assumption). Figure 2.4 depicts the action by four processors on a 32×8 matrix.

For this example, each processor performs 28 rotations *independently* and *simultaneously*. In the IAP, each processor does equal work, and there is no idle time or synchronization cost.

2.2.2 The Cooperative Annihilation Phase

In the CAP, the rows in the p upper triangular submatrices are assigned to groups by collecting the rows with leading nonzero in column j into a group G_j , $1 \leq j \leq n$. For the case $m/p \geq n$, at the end of the IAP we have exactly p rows in each G_j for $1 \leq j \leq n$. The collection of rows in G_j can be viewed as a $p \times (n - j + 1)$ rectangular submatrix.

$$\begin{pmatrix}
\times & \times & \times & \times & \times & \times & \times & \times \\
1 & \times & \times & \times & \times & \times & \times & \times \\
2 & 3 & \times & \times & \times & \times & \times & \times \\
4 & 5 & 6 & \times & \times & \times & \times & \times \\
7 & 8 & 9 & 10 & \times & \times & \times & \times \\
11 & 12 & 13 & 14 & 15 & \times & \times & \times \\
16 & 17 & 18 & 19 & 20 & 21 & \times & \times \\
22 & 23 & 24 & 25 & 26 & 27 & 28 & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
1 & \times & \times & \times & \times & \times & \times & \times \\
2 & 3 & \times & \times & \times & \times & \times & \times \\
4 & 5 & 6 & \times & \times & \times & \times & \times \\
7 & 8 & 9 & 10 & \times & \times & \times & \times \\
11 & 12 & 13 & 14 & 15 & \times & \times & \times \\
16 & 17 & 18 & 19 & 20 & 21 & \times & \times \\
22 & 23 & 24 & 25 & 26 & 27 & 28 & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
1 & \times & \times & \times & \times & \times & \times & \times \\
2 & 3 & \times & \times & \times & \times & \times & \times \\
4 & 5 & 6 & \times & \times & \times & \times & \times \\
7 & 8 & 9 & 10 & \times & \times & \times & \times \\
11 & 12 & 13 & 14 & 15 & \times & \times & \times \\
16 & 17 & 18 & 19 & 20 & 21 & \times & \times \\
22 & 23 & 24 & 25 & 26 & 27 & 28 & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
1 & \times & \times & \times & \times & \times & \times & \times \\
2 & 3 & \times & \times & \times & \times & \times & \times \\
4 & 5 & 6 & \times & \times & \times & \times & \times \\
7 & 8 & 9 & 10 & \times & \times & \times & \times \\
11 & 12 & 13 & 14 & 15 & \times & \times & \times \\
16 & 17 & 18 & 19 & 20 & 21 & \times & \times \\
22 & 23 & 24 & 25 & 26 & 27 & 28 & \times
\end{pmatrix}$$

Figure 2.4: Independent annihilation by four processors.

If a Givens rotation is applied to the i^{th} row and the j^{th} row to annihilate the leading nonzero $a_{j,k}$ in the j^{th} row, row i is referred to as the "pivot row". By choosing one row in each group as the pivot row, the task of eliminating the leading nonzeros in the remaining $(p-1)$ rows in one group is *independent* of the same task in another group. We shall arbitrarily choose the lowest numbered row in each group as the pivot row. In order to have these independent tasks performed by the p processors simultaneously, and also maintain the work load balance, we assign groups G_1 to G_p to the p processors in order, with the assignment of group G_{p+1} "wrapping around" to processor 1. Figure 2.5 illustrates the *initial* data assignment for the example in Figure 2.4. The matrix elements assigned to the i^{th} processor are labelled by P_i.

The "top" submatrix in Figure 2.5 contains the pivot rows, and the main diagonals of the remaining $(p-1)$ submatrices correspond to the nonzero elements which can be eliminated by the p processors independently and simultaneously. Recall that using the wrap mapping, each processor is assigned n/p groups, and each group has $(p-1)$ leading nonzeros to be eliminated. Therefore, after the p processors each perform $n(p-1)/p$ rotations (in parallel), we have eliminated the $(p-1)$ main diagonals of the "bottom" $(p-1)$ triangular submatrices. The remaining nonzeros are depicted in Figure 2.6. We then apply the same idea to eliminate the elements along the first superdiagonal in each of the $(p-1)$ submatrices, using the rows in the top submatrix as the pivot rows.

The elimination of the diagonals can be implemented in several ways. In the implementation we describe in the next section, the pivot rows in the top submatrix are statically assigned to the p processors using a wrap mapping as explained earlier, and the remaining data are accessed in groups by each processor. To be accurate in what follows, we redefine group G_j initially to contain $(p-1)$ rows with leading nonzero in column j excluding the j^{th} pivot row. We adopt the convention of labelling the elements of pivot rows by P_i if they are assigned to processor P_i in the static mapping, and the remaining rows are labelled by their

$$\begin{pmatrix}
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
 & & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
 & & & P_4 & P_4 & P_4 & P_4 & P_4 \\
 & & & & P_1 & P_1 & P_1 & P_1 \\
 & & & & & P_2 & P_2 & P_2 \\
 & & & & & & P_3 & P_3 \\
 & & & & & & & P_4 \\
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
 & & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
 & & & P_4 & P_4 & P_4 & P_4 & P_4 \\
 & & & & P_1 & P_1 & P_1 & P_1 \\
 & & & & & P_2 & P_2 & P_2 \\
 & & & & & & P_3 & P_3 \\
 & & & & & & & P_4 \\
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
 & & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
 & & & P_4 & P_4 & P_4 & P_4 & P_4 \\
 & & & & P_1 & P_1 & P_1 & P_1 \\
 & & & & & P_2 & P_2 & P_2 \\
 & & & & & & P_3 & P_3 \\
 & & & & & & & P_4 \\
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
 & & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
 & & & P_4 & P_4 & P_4 & P_4 & P_4 \\
 & & & & P_1 & P_1 & P_1 & P_1 \\
 & & & & & P_2 & P_2 & P_2 \\
 & & & & & & P_3 & P_3 \\
 & & & & & & & P_4
\end{pmatrix}$$

Figure 2.5: CAP–initial data distribution among four processors.

$$\begin{pmatrix}
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
 & & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
 & & & P_4 & P_4 & P_4 & P_4 & P_4 \\
 & & & & P_1 & P_1 & P_1 & P_1 \\
 & & & & & P_2 & P_2 & P_2 \\
 & & & & & & P_3 & P_3 \\
 & & & & & & & P_4 \\
0 & x & x & x & x & x & x & x \\
 & 0 & x & x & x & x & x & x \\
 & & 0 & x & x & x & x & x \\
 & & & 0 & x & x & x & x \\
 & & & & 0 & x & x & x \\
 & & & & & 0 & x & x \\
 & & & & & & 0 & x \\
 & & & & & & & 0 \\
0 & x & x & x & x & x & x & x \\
 & 0 & x & x & x & x & x & x \\
 & & 0 & x & x & x & x & x \\
 & & & 0 & x & x & x & x \\
 & & & & 0 & x & x & x \\
 & & & & & 0 & x & x \\
 & & & & & & 0 & x \\
 & & & & & & & 0 \\
0 & x & x & x & x & x & x & x \\
 & 0 & x & x & x & x & x & x \\
 & & 0 & x & x & x & x & x \\
 & & & 0 & x & x & x & x \\
 & & & & 0 & x & x & x \\
 & & & & & 0 & x & x \\
 & & & & & & 0 & x \\
 & & & & & & & 0
\end{pmatrix}$$

Figure 2.6: CAP–after the annihilation of three main diagonals.

respective group number G_j. Referring to Figure 2.7, we see that after the annihilation of main diagonals, the leading nonzero position in group G_j becomes $(j+1)$. Therefore, the processor which is assigned the j^{th} $(j > 1)$ pivot row will now access group G_{j-1} to eliminate the current leading nonzero elements in column j. We illustrate this mapping in Figure 2.7, where the three superdiagonals correspond to the elements to be eliminated in this step.

After the elements on the first superdiagonals are eliminated, the processor which is assigned the j^{th} $(j > 2)$ pivot row can now annihilate the current leading nonzero elements in group G_{j-2}. The elements to be eliminated in this step lie on the second superdiagonals. Thus, it is clear that if the p processors simply synchronize with each other before starting annihilation of elements along each diagonal, the $(p-1)$ submatrices are eliminated one diagonal at a time without any other synchronization cost. In particular, all processors access disjoint sets of pivot rows and disjoint groups when eliminating elements along the same diagonal. Thus, there are *no* shared data. Since each $n \times n$ upper triangular submatrix has n diagonals, the synchronization cost for the parallel algorithm is clearly $O(n)$. Such an implementation is particularly suitable for a machine with *high* synchronization overhead. We refer to this version of implementation as the *synchronous* implementation. In the next section we discuss some implementation details and describe an *asynchronous* implementation which can further reduce the processor idle time by increasing the synchronization cost to $O(n^2/p)$.

2.3 Implementation Issues

The implementation of the independent annihilation phase (IAP) is straightforward. We therefore concentrate on the implementation of the cooperative annihilation phase (CAP) in this section. The data accessed by each individual processor during the entire elimination process is *dictated* by the initial static allocation of pivot rows. For example, if processor

$$\begin{pmatrix}
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
 & & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
 & & & P_4 & P_4 & P_4 & P_4 & P_4 \\
 & & & & P_1 & P_1 & P_1 & P_1 \\
 & & & & & P_2 & P_2 & P_2 \\
 & & & & & & P_3 & P_3 \\
 & & & & & & & P_4 \\
0 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 \\
 & 0 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 \\
 & & 0 & G_3 & G_3 & G_3 & G_3 & G_3 \\
 & & & 0 & G_4 & G_4 & G_4 & G_4 \\
 & & & & 0 & G_5 & G_5 & G_5 \\
 & & & & & 0 & G_6 & G_6 \\
 & & & & & & 0 & G_7 \\
 & & & & & & & 0 \\
0 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 \\
 & 0 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 \\
 & & 0 & G_3 & G_3 & G_3 & G_3 & G_3 \\
 & & & 0 & G_4 & G_4 & G_4 & G_4 \\
 & & & & 0 & G_5 & G_5 & G_5 \\
 & & & & & 0 & G_6 & G_6 \\
 & & & & & & 0 & G_7 \\
 & & & & & & & 0 \\
0 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 \\
 & 0 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 \\
 & & 0 & G_3 & G_3 & G_3 & G_3 & G_3 \\
 & & & 0 & G_4 & G_4 & G_4 & G_4 \\
 & & & & 0 & G_5 & G_5 & G_5 \\
 & & & & & 0 & G_6 & G_6 \\
 & & & & & & 0 & G_7 \\
 & & & & & & & 0
\end{pmatrix}$$

Figure 2.7: CAP–data mapping to eliminate the first superdiagonals.

P_i is assigned pivot rows

$$\{k_1, k_2, \ldots, k_{\mu_i}\},$$

processor P_i participates in eliminating the elements along the main diagonal by accessing data in groups

$$\{G_{k_1}, G_{k_2}, \ldots, G_{k_{\mu_i}}\}.$$

To eliminate elements along the j^{th} superdiagonal, processor P_i accesses data in groups

$$\{G_{k_1-j}, G_{k_2-j}, \ldots, G_{k_{\mu_i}-j}\}.$$

Encountering a group G_ρ with $\rho \leq 0$ simply indicates that there are no more rows with leading nonzero in the same position as the corresponding pivot row. For example, if $(k_1 - j) \leq 0$, the pivot row k_1 is not be modified any more and is row k_1 of the upper triangular factor R.

Observe that the group G_ρ must be eliminated against pivot rows ρ, $\rho + 1$, ..., n in strict order. Whether G_ρ can be reduced against pivot row $(\rho + \eta)$ can be determined by simply checking whether the current position of its leading nonzero elements is in column $(\rho + \eta)$. Therefore, if we associate with each group G_j a shared variable *first*$[j]$ to indicate the current position of its leading nonzeros, all processors can proceed by themselves to complete their share of work in the entire CAP process using the following synchronization mechanism. For convenience in describing the algorithm, we assume that processor P_i is assigned pivot rows $\{k_1, k_2, \cdots, k_{\mu_i}\}$. The pivot row numbers for P_i are stored in a local array *pvts*$[j], 1 \leq j \leq \mu_i$. We also need a global array *first* to record the current position of the leading nonzeros for each group, and a local array *map* to identify the groups which are currently due to be processed by each individual processor. Note that *first* is shared among multiple processors. Therefore, our implementation must ensure that only one processor can update *first* at a time. The basic algorithm executed by processor P_i can now be expressed in the following form.

```
for $j = 1, 2, \ldots, \mu_i$ do
    $pvts[j] \leftarrow k_j$
    $map[j] \leftarrow k_j$
$jstrt \leftarrow 1$
while $jstrt \leq \mu_i$ do
    for $j = jstrt, jstrt + 1, \ldots, \mu_i$ do
        $\rho \leftarrow map[j]$
        wait until $first[\rho] = pvts[j]$
        reduce rows in group $G_\rho$ using pivot row $pvts[j]$
        $first[\rho] \leftarrow first[\rho] + 1$
        $map[j] \leftarrow map[j] - 1$
    if $map[jstrt] = 0$ then
        $jstrt \leftarrow jstrt + 1$
```

The algorithm and its implementation can easily handle the case where $m/p < n$. For $p = 4$, a matrix of order 16×8 is such an example. For this example, in the IAP each processor will reduce its block of rows to an m/p by n upper trapezoidal submatrix. The number of rotations performed by each processor is $(m/p - 1)m/2p$. The remaining nonzero elements are shown in Figure 2.8 for an 16×8 example.

When $m/p < n$, the top upper trapezoidal submatrix does not contain a full set of pivot rows. For the example in Figure 2.8, the pivot rows 5, 6, 7, and 8 are all to be generated during the elimination process in the CAP. An important observation which facilitates a clean implementation is that the currently non-existent pivot rows will each be generated from the data in an existing group. Therefore, we can still statically assign n pivot rows to the p processors and record them in the *pvts* and *map* arrays as before. In addition, each processor records whether group G_i exists for $1 \leq i \leq n$. For each entry in $map[i]$,

$$\begin{pmatrix}
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
 & & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
 & & & P_4 & P_4 & P_4 & P_4 & P_4 \\
G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 \\
 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 \\
 & & G_3 & G_3 & G_3 & G_3 & G_3 & G_3 \\
 & & & G_4 & G_4 & G_4 & G_4 & G_4 \\
G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 \\
 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 \\
 & & G_3 & G_3 & G_3 & G_3 & G_3 & G_3 \\
 & & & G_4 & G_4 & G_4 & G_4 & G_4 \\
G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 & G_1 \\
 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 & G_2 \\
 & & G_3 & G_3 & G_3 & G_3 & G_3 & G_3 \\
 & & & G_4 & G_4 & G_4 & G_4 & G_4
\end{pmatrix}$$

Figure 2.8: End of IAP – a 16×8 example.

the processor processes $G_{map[i]}$ if it exists, and does nothing other than updating *map*[i] to be (*map*[i] $-$ 1) if $G_{map[i]}$ does not exist. Since *map*[i] is updated each time, eventually $G_{map[i]}$ refers to existing data with leading nonzero in position *pvts*[i]. One row is now taken from $G_{map[i]}$ to become the pivot row, and the remaining rows (if there are any) are further reduced.

2.4 Analysis of Synchronization Cost

The actual implementation of the synchronization mechanism for the general case ($m/p < n$ or $m/p \geq n$) needs to maintain two attributes for each group G_i, namely *first*[i], the current position of the leading nonzero elements in G_i, and *nrows*[i], the current number of rows in G_i. Both items are shared data to be updated and read by multiple processors. The exclusive access of these two items is ensured by a lock associated with group G_i, namely *gplock*[i]. We let β denote the total time of *acquiring* and *releasing* a lock. The synchronization mechanism described in the last section requires β time for each updating of the array *first*. For the general case, *nrows*[i] can be updated together with *first*[i] under the same lock.

We now analyze the synchronization cost for the cases $m/p \geq n$ and $m/p < n$ separately. For convenience, we assume m and n are integral multiples of p. If $m/p \geq n$, we have n pivot rows and n groups of data. Since the i^{th} pivot row is used to eliminate the leading nonzero elements in groups G_i, G_{i-1}, ..., and G_1 in strict order, there are exactly i groups to be processed by the i^{th} pivot row. The synchronization cost associated with the i^{th} pivot row is therefore $i\beta$. Recalling that the pivot rows are assigned to the p processors using a wrap mapping, we can therefore compute the synchronization cost of processor P_k by

$$S(P_k) \quad = \quad \beta \sum_{\ell=0}^{\frac{n}{p}-1} (k + p\ell)$$

$$= \beta\left(\frac{n^2}{2p} - \frac{n}{2} + k\frac{n}{p}\right), \tag{2.1}$$

where $1 \le k \le p$. For the case $m/p < n$, the synchronization cost associated with the first m/p pivot rows is $i\beta$, where $1 \le i \le m/p$. For $m/p < j \le n$, the synchronization cost associated with row j is at most $\beta m/p$. Assuming further that m/p is also an integral multiple of p, the synchronization cost of processor P_k is as shown in (2.2).

$$\begin{aligned} S(P_k) &\le \beta\left(\sum_{\ell=0}^{\frac{m}{p^2}-1}(k+p\ell)\right) + \beta\left(\frac{m}{p}\frac{(n-m/p)}{p}\right) \\ &= \beta\left(\frac{mn}{p^2} - \frac{m^2}{2p^3} + k\frac{m}{p^2} - \frac{m}{2p}\right). \end{aligned} \tag{2.2}$$

2.5 Analysis of Work Load Distribution

We now examine how the computational work is distributed among the p processors in the CAP. We denote the work performed by the k^{th} processor by $W_p(P_k)$, $1 \le k \le p$, and first consider the case $m/p \ge n$. Processor P_k is assigned the set of pivot rows $\{k, k+p, k+2p, \cdots, n-k+p\}$. As shown in the last section, the number of groups to be processed by the i^{th} pivot row is exactly i, and there are $(p-1)$ rows in each group. The elimination of one of the $(p-1)$ leading nonzeros in column i requires one rotation applied to a pair of rows of length $(n-i+1)$, which amounts to $4(n-i+1)$ multiplicative operations. Thus,

$$\begin{aligned} W_p(P_k) &= 4(p-1)\sum_{\ell=0}^{\frac{n}{p}-1}(k+\ell p)(n-k-\ell p+1) \\ &= \frac{(p-1)}{p}\left(\frac{2}{3}n^3 + 2n^2 - n\left(2p + \frac{2}{3}p^2 - 4k - 4kp + 4k^2\right)\right), \end{aligned} \tag{2.3}$$

where $1 \le k \le p$. The total serial work in the CAP can be computed by assuming that the $(p-1)$ upper triangular submatrices are eliminated by a single processor one submatrix at a time, using the rows in the top submatrix as pivot rows. This yields

$$\begin{aligned} W_1 &= 4(p-1)\sum_{i=1}^{n}\frac{i(i+1)}{2} \\ &= (p-1)\left(\frac{2}{3}n^3+2n^2+\frac{4}{3}n\right). \end{aligned} \tag{2.4}$$

The optimal work load distribution is thus

$$\frac{W_1}{p} = \frac{(p-1)}{p}\left(\frac{2}{3}n^3+2n^2+\frac{4}{3}n\right). \tag{2.5}$$

Comparing equations (2.3) and (2.5), we see that the work distribution of the CAP differs from the optimal distribution in only the low order $O(n)$ terms.

For the case $m/p < n$, we further assume that m/p is an integral multiple of p, and that $n = s\frac{m}{p}$, where s is an integer in $[\,2,\ p\,]$. The following observations are useful in deriving the work load of processor P_k.

Observation 1. $\frac{m}{p} < n$ and $n = s\frac{m}{p}$ imply that $\frac{m}{p}$ pivot rows exist at the beginning of the CAP, and that the other $(s-1)\frac{m}{p}$ pivot rows are to be taken from the remaining $\frac{m}{p}(p-1)$ rows during the CAP.

Observation 2. The wrap mapping dictates that pivot rows

$$\begin{aligned} \{\ & \tfrac{m}{p}+k, \tfrac{m}{p}+p+k, \cdots, \tfrac{m}{p}+(\tfrac{m}{p^2}-1)p+k, \\ & 2\tfrac{m}{p}+k, 2\tfrac{m}{p}+p+k, \cdots, 2\tfrac{m}{p}+(\tfrac{m}{p^2}-1)p+k, \\ & \qquad\vdots \\ & (s-1)\tfrac{m}{p}+k, (s-1)\tfrac{m}{p}+p+k, \cdots, (s-1)\tfrac{m}{p}+(\tfrac{m}{p^2}-1)p+k\ \ \} \end{aligned}$$

are to be generated by processor P_k.

Observation 3. The total number of rows to be reduced by processor P_k by pivot row $\left(i\frac{m}{p}+\ell p+k\right)$ is $\left(\frac{m}{p}(p-i)-\ell p-k\right)$, and the length of each row is $\left(n-i\frac{m}{p}-\ell p-k+1\right)$.

Applying these observations, we obtain

$$\begin{aligned} W_p(P_k) &= 4(p-1)\sum_{\ell=0}^{\frac{m}{p^2}-1}(k+p\ell)(n-k-p\ell+1)+ \\ &\quad 4\sum_{i=1}^{s-1}\sum_{\ell=0}^{m/p^2-1}\left(\frac{m}{p}(p-i)-\ell p-k\right)\left(n-i\frac{m}{p}-\ell p-k+1\right) \\ &= \left(2s^2-2s+\frac{2}{3}\right)\frac{m^3}{p^3}-\frac{2}{3}s^3\frac{m^3}{p^4}+(4s-2)\frac{m^2}{p^2}-2s^2\frac{m^2}{p^3} \\ &\quad +O(m)\,. \end{aligned} \tag{2.6}$$

The total CAP serial work for the case $m/p < n$ can be computed by subtracting the total IAP serial work from the total work:

$$\begin{aligned} W_1 &= 4\sum_{i=1}^{n}(m-i)(n-i+1)-4\sum_{i=1}^{\frac{m}{p}-1}p\left(\frac{m}{p}-i\right)(n-i+1) \\ &= 2mn^2-\frac{2}{3}n^3-2\frac{m^2n}{p}+4mn-2n^2+\frac{2}{3}\frac{m^3}{p^2}-2\frac{m^2}{p} \\ &\quad +\frac{4}{3}(m-n)\,. \end{aligned} \tag{2.7}$$

The optimal work load distribution is W_1/p. In order to compare with $W_p(P_k)$, we simplify W_1/p by substituting $n = s\frac{m}{p}$. This yields

$$\begin{aligned} \frac{W_1}{p} &= \left(2s^2-2s+\frac{2}{3}\right)\frac{m^3}{p^3}-\frac{2}{3}s^3\frac{m^3}{p^4}+(4s-2)\frac{m^2}{p^2}-2s^2\frac{m^2}{p^3} \\ &\quad +O(m)\,. \end{aligned} \tag{2.8}$$

Comparing $W_p(P_k)$ with W_1/p shows that

$$W_p(P_k)-\frac{W_1}{p} = O(m)\,. \tag{2.9}$$

Thus, for both cases ($m/p < n$, $m/p \geq n$), $W_p(P_k)$ and W_1/p differ only in their low order terms.

2.6 Performance Analysis

To analyze the performance of the parallel algorithm in the CAP, first note that the nonzero elements in the same group are eliminated by *different* processors in strict order. It is clear that when p groups of data are processed in parallel, the time is bounded by the processing time of the group which requires the largest amount of computation. We consider the parallel time (in units of multiplicative operations) for factoring an $m \times n$ matrix using p processors. As usual we assume m and n are integral multiples of p.

We first consider the case $m/p \geq n$. In this case, there are n groups of data, each of $(p-1)$ rows, to be eliminated entirely in the CAP. Because the n groups are assigned to the p processors using a wrap mapping, an upper bound of the CAP parallel time T_p is given by assuming that the nonzeros in groups $G_1, G_{p+1}, G_{2p+1}, \ldots, G_{n-p+1}$ are eliminated sequentially. Letting $\Lambda(T_p)$ denote the upper bound of T_p, we obtain

$$\begin{aligned}\Lambda(T_p) &= 4(p-1)\sum_{\ell=0}^{\frac{n}{p}-1}\frac{1}{2}(n-p\ell)(n-p\ell+1)\\ &= \frac{(p-1)}{p}\left(\frac{2}{3}n^3\right)+pn^2-\frac{n^2}{p}+O(n)\,. \qquad (2.10)\end{aligned}$$

Note that $T_p = \Lambda(T_p)$ for the synchronous implementation of the CAP.

For the asynchronous implementation described in detail in Section 3, when P_α is processing group G_{n-p+1} to eliminate its leading nonzeros in the j^{th} column, it is possible for a different processor P_γ to start its processing of group G_1 to eliminate its leading nonzeros in the $(j+1)^{st}$ position. We thus expect $T_p < \Lambda(T_p)$. Nevertheless, when $m/p \geq n$, the serial time for the CAP is given by

$$T_1 = (p-1)\left(\frac{2}{3}n^3\right)+O(n^2)\,,$$

and $\Lambda(T_p)$ is thus *optimal* in its leading term. We therefore cannot expect dramatic improvement in T_p using the asynchronous version of the implementation. Experimental results given in Section 2.7 confirm this expectation.

For the case $m/p < n$, we again assume that m/p is an integral multiple of p, and that $n = s\frac{m}{p}$, where s is an integer in $[2, p]$. The following observations are helpful in analyzing the performance of the algorithm.

Observation 1. $\frac{m}{p} < n$ and $n = s\frac{m}{p}$ imply that $(s-1)\frac{m}{p}$ pivot rows are absent at the beginning of the CAP.

Observation 2. Since processor P_k is assigned pivot rows

$$\left\{k, k+p, \cdots, \frac{m}{p} - p + k\right\},$$

the $(p-1)$ rows in group $G_{\frac{m}{p}}$ will initially be reduced by processor P_p and have the position of their leading nonzeros updated to be $first\left(\frac{m}{p}\right) = \frac{m}{p} + 1$. Because pivot row $\left(\frac{m}{p} + 1\right)$ does not exist, when group $G_{\frac{m}{p}}$ is next processed by processor P_1, one row will be taken to serve as the pivot row. In general, the wrap mapping dictates that the first $(p-1)$ pivot rows are taken from group $G_{\frac{m}{p}}$, and the next $(p-1)$ pivot rows are taken from group $G_{\frac{m}{p}-1}$, and so on, until all of the pivot rows are present.

Observation 3. Based on observation 2, an upper bound for the parallel time of the CAP is the total time to process the groups G_1, G_{p+1}, G_{2p+1}, $\cdots$ and $G_{\frac{m}{p}-p+1}$ sequentially. The number of rows in each group could change dynamically during the CAP, and hence we have to keep track of the actual number of rows in our analysis in order to have a sufficiently tight upper bound.

The following lemmas are needed to prove Theorem 2.4.

Lemma 2.1 *If all of the n pivot rows are already present, then the total number of multiplicative operations required to eliminate the nonzeros in groups G_1, G_{p+1}, G_{2p+1}, $\cdots$ and $G_{\frac{m}{p}-p+1}$ is given by*

$$\hat{T}_p = 4(p-1)\sum_{\ell=0}^{\frac{m}{p^2}-1} \frac{1}{2}(n-p\ell)(n-p\ell+1)$$

$$\begin{aligned}
&= 2\frac{mn^2}{p} - 2\frac{m^2n}{p^2} + \frac{2}{3}\frac{m^3}{p^3} + 2mn + \frac{2}{3}m - \frac{m^2}{p} + \frac{1}{3}pm - \\
&\quad 2\frac{n^2m}{p^2} - 2\frac{mn}{p^2} + 2\frac{m^2n}{p^3} + \frac{m^2}{p^3} - \frac{m}{p} - \frac{2}{3}\frac{m^3}{p^4} \\
&= (2s^2 - 2s + \frac{2}{3})\frac{m^3}{p^3} - (2s^2 - 2s + \frac{2}{3})\frac{m^3}{p^4} + O\left(m^2\right). \qquad (2.11)
\end{aligned}$$

Lemma 2.2 *If we assume for convenience that $\mu = \frac{(s-1)\frac{m}{p}}{p(p-1)}$ is an integer, then $(\mu-1)(p-1)$ pivot rows will be taken from groups $G_{\frac{m}{p}-p+1}$, $G_{\frac{m}{p}-2p+1}$, $\cdots$, and $G_{\frac{m}{p}-(\mu-1)p+1}$ in order. Moreover, the size of the ℓ^{th} row taken from group $G_{\frac{m}{p}-ip+1}$ is given by*

$$f(m,n,i,p,\ell) = n - \frac{m}{p} - (p-1)^2 - (i-1)(p-1)p - \ell + 1 .$$

Proof: This follows directly from observation 2. □

Lemma 2.3 *The number of multiplicative operations to be saved by not eliminating a row of size η is $\eta(\eta+1)/2$.*

Theorem 2.4 *An upper bound for the parallel time for the CAP is given by*

$$\Lambda(T_p) = \left(2s^2 - 2s + \frac{2}{3}\right)\frac{m^3}{p^3} - \frac{2}{3}s^3\frac{m^3}{p^4} + O\left(m^2\right).$$

Proof: From Lemmas 2.1 to 2.3, we have

$$\begin{aligned}
\Lambda(T_p) &= \hat{T}_p - 4\sum_{i=1}^{\mu-1}\sum_{\ell=1}^{p-1}\frac{1}{2}f(m,n,i,p,\ell)\left(f(m,n,i,p,\ell)+1\right), \\
&= \left(2s^2 - 2s + \frac{2}{3}\right)\frac{m^3}{p^3} - \frac{2}{3}s^3\frac{m^3}{p^4} + O\left(m^2\right). \qquad (2.12)
\end{aligned}$$

□

The total CAP serial work W_1 for the case $m/p < n$ was derived in the last section. We use it here to represent the serial time T_1. Comparing $\Lambda(T_p)$ in Theorem 4 with T_1/p from equation (8), we have

$$\Lambda(T_p) - \frac{T_1}{p} = O\left(m^2\right). \qquad (2.13)$$

Since $T_p \leq \Lambda(T_p)$, we have shown again that the CAP parallel time T_p is *optimal* in its leading term. The actual performance of the algorithm is reported in the next section.

We now explain why $\Lambda(T_p)$ has included the time a processor may spend waiting for a lock to be unlocked. Recall that the p processors take turns processing each group of data. The lock associated with each group is thus accessed by two different processors in any two consecutive times. Let us consider any one particular group in G_1, G_{p+1}, G_{2p+1}, $\ldots$, G_{n-p+1}, say G_α, and suppose that processor P_γ is currently processing this group. We can immediately infer from our data mapping strategy that the next processor to process group G_α is P_ξ, where $\xi = \gamma + 1$ if $\gamma < p$, or $\xi = 1$ if $\gamma = p$. To determine the time P_ξ may be waiting on the lock associated with G_α, recall that in computing $\Lambda(T_p)$ we assume that the nonzeros in groups G_1, G_{p+1}, G_{2p+1}, $\ldots$, G_{n-p+1} are eliminated sequentially. This assumption implies that P_ξ waits until group G_α is completely processed by P_γ, i.e., the maximum possible waiting time has been included in $\Lambda(T_p)$.

2.7 Numerical Experiments

Our experiments were performed on a Sequent Balance 8000 parallel computer having 8 identical processors and 8M bytes of global memory. All processors and memory modules are linked by a high speed bus. The parallel algorithms described in this chapter were implemented in FORTRAN. Since the use of Givens rotations is row-oriented and a two-dimensional array is stored column by column in FORTRAN, the transpose of the coefficient matrix was stored and operations on the matrix were done in a column-oriented manner. The execution times reported below exclude only the time for generating the coefficient matrix. In particular, for the parallel algorithm, the execution time includes the overhead in creating multiple processes. Since spawning a child process on the Balance 8000 is an expensive operation (30-50 milliseconds), this overhead is significant for small problems and large numbers of processes.

The execution times (in seconds) of the serial and parallel algorithms are denoted by T_s and T respectively, and as in previous sections, p denotes the number of processors. T_s and T should not be confused with T_1 and T_p in our performance analysis section, because the latter two represent the serial and parallel time for the CAP only.

Table 2.1 gives some timing results of the serial algorithm and the asynchronous implementation of the parallel algorithm. The entries of the $m \times n$ test matrices were generated using a random number generator. The reported efficiency is computed using

$$\textit{efficiency} = \frac{T_s}{p \times T} .$$

Table 2.2 compares the execution time of the asynchronous implementation with the synchronous implementation for the same set of matrices. The results show that the former is slightly faster than the latter for all test matrices. This is not surprising because the synchronization overhead on the Balance 8000 is very low, so the saving from reducing synchronization cost is evidently less significant than the saving from reducing idle time. However, as our analysis in Section 2.6 predicted, the difference is not very great.

To study the effect of high synchronization cost, we simulated that situation by executing a dummy assignment statement five hundred times whenever a lock was accessed. We then compared the asynchronous implementation, the synchronous implementation, and our implementation of the pipelined Givens method given in [13] for problems of three different sizes. For the 1000×100 matrix, since $m \gg n$, the effect of high synchronization cost would be expected to be most dramatic for the pipelined Givens method, since its synchronization cost can be shown to be $O(mn/p)$. The difference between the pipelined Givens method and the asynchronous Givens method would be expected to diminish as m approaches n. Table 2.3 supports these expectations. The synchronous Givens algorithm has the lowest synchronization cost, and performs best among the three when the synchronization cost is high and $m \gg n$.

m	n	p	T_s	T	*efficiency*
100	100	1	47.17		
		2		24.07	98%
		3		16.25	97%
		4		12.13	97%
		5		9.82	96%
		6		8.47	93%
		7		7.63	88%
200	200	1	368.95		
		2		187.80	98%
		3		125.60	98%
		4		93.90	98%
		5		75.20	98%
		6		63.60	97%
		7		54.70	96%
200	40	1	22.80		
		2		11.63	98%
		3		7.87	97%
		4		5.98	95%
		5		4.85	94%
		6		4.18	91%
		7		3.68	89%
500	100	1	336.30		
		2		168.90	99.6%
		3		113.00	99.3%
		4		84.80	99.3%
		5		67.90	99.0%
		6		57.27	97.8%
		7		49.30	97.5%

Table 2.1: Execution time on the Balance 8000.

m	n	p	Asynchronous T	Synchronous T
100	100	2	24.07	24.10
		3	16.25	16.73
		4	12.13	12.87
		5	9.82	10.87
		6	8.47	9.82
		7	7.63	8.88
200	200	2	187.80	188.42
		3	125.60	127.07
		4	93.90	96.88
		5	75.20	79.38
		6	63.60	69.13
		7	54.70	61.50
200	40	2	11.63	11.65
		3	7.87	7.97
		4	5.98	6.07
		5	4.85	5.03
		6	4.18	4.48
		7	3.68	3.95
500	100	2	168.90	168.70
		3	113.00	113.27
		4	84.80	85.37
		5	67.90	69.02
		6	57.27	58.58
		7	49.30	51.62

Table 2.2: Execution time on the Balance 8000.

$m = 1000, n = 100$, $p = 5$, $T_s = 696.8$ sec		
Parallel Algorithm	Execution Time	Efficiency
Pipelined Givens	211.80 sec	66%
Asynchronous Givens	144.03 sec	97%
Synchronous Givens	141.77 sec	98%
$m = 500, n = 100$, $p = 5$, $T_s = 336.3$ sec		
Parallel Algorithm	Execution Time	Efficiency
Pipelined Givens	101.57 sec	66%
Asynchronous Givens	71.58 sec	94%
Synchronous Givens	70.52 sec	98%
$m = 100, n = 100$, $p = 5$, $T_s = 47.2$ sec		
Parallel Algorithm	Execution Time	Efficiency
Pipelined Givens	13.87 sec	68%
Asynchronous Givens	10.52 sec	90%
Synchronous Givens	11.60 sec	81%

Table 2.3: The effect of high synchronization cost.

2.8 Concluding Remarks

The algorithm we propose in this chapter is similar in spirit to Sameh's 2-stage algorithm described in [76]. In Sameh's brief survey article, he suggests a parallel algorithm to realize orthogonal factorization on a multiprocessor consisting of clusters of tightly-coupled processors. The multiprocessor model he considers has the form shown in Figure 2.9, where,

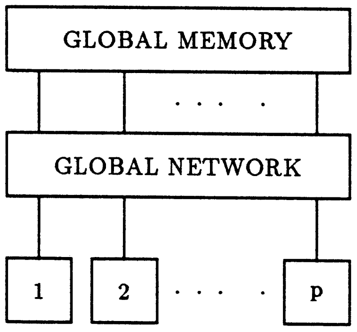


Figure 2.9: A multiprocessor model considered in [76].

for factoring an $m \times n$ matrix, he assumes that (i) each of the p units in the multiprocessor model is a cluster consisting of n processors interconnected in the form of a tree, and (ii) the communication cost between one cluster and another far exceeds that between two processors in one cluster.

The algorithm proposed in [76] consists of two stages. In the first stage, A is partitioned as $A^T = \left(A_1^T, A_2^T, \ldots, A_p^T\right)$. Each cluster k (tree of n processors) obtains the orthogonal factorization $Q_k A_k = \left(R_k^T, O\right)^T$, where each R_k is either upper triangular or upper trapezoidal. The first stage is thus similar to the independent annihilation phase of our algorithm.

The second stage of Sameh's algorithm is different from our algorithm. The differences are necessary because the two algorithms are targeted for very different architectures. The multiprocessor considered in [76] is subject to the assumption that each cluster has its local memory and the data movement between clusters is very expensive. Due to such concern, the computational work in the second stage of Sameh's algorithm is performed within each cluster with respect to its local data. Sameh describes the second stage for the case $m/p > n$ only. In this case each R_k is upper triangular. To allow cluster k to eliminate R_k, $2 \leq k \leq p$, locally, the rows of R_1 serve as pivot rows and they are transmitted to the $p-1$ other clusters in a "pipelined" manner. Consequently, the n processors in cluster 1 will not share the computational work in the second stage.

However, in our multiprocessor model, all data are stored in the common memory accessible to all p processors via a high speed bus. Our goal is to have all p processors gainfully employed throughout the entire computing process. Therefore, in our design of the cooperative annihilation phase, each processor not only works on data different from the first phase, the data each processor is assigned also change dynamically during the CAP. In addition, our algorithm and its implementation are general including the cases $m/p \geq n$ and $m/p < n$. Our analysis and numerical experiments indicate that our goal is achieved in both cases.

Chapter 3

QR Factorization of a Dense Matrix on a Hypercube Multiprocessor

3.1 Introduction

In this chapter we present an algorithm for factoring an $m \times n$ matrix using orthogonal transformations on a hypercube multiprocessor. When $m \geq n$, the mathematical computation we consider is usually formulated as

$$QA = \begin{pmatrix} R \\ 0 \end{pmatrix} ,$$

where A is an $m \times n$ matrix with full column rank, Q is an $m \times m$ orthogonal matrix and R is an upper triangular matrix of order n. When $m < n$, we have

$$QA = (R\, S) ,$$

where A is an $m \times n$ matrix with full row rank, Q is as defined above, R is an upper triangular matrix of order m, and S is an $m \times (n-m)$ rectangular matrix.

Our scheme involves the embedding of a two-dimensional grid in the hypercube network. For easy exposition, we first describe a special case of the algorithm in order to explain some basic strategies for data mapping and inter-processor communication on the hypercube. We refer to the special case as Algorithm I, and the general algorithm as Algorithm II. Finally we propose a further enhancement to reduce both arithmetic and communication costs of Algorithm II. This version of the algorithm is referred to as the enhanced Algorithm II.

Algorithms I and II reduce A to R or $(R\ S)$ by forming Q as the product of Givens rotations. Since there is much freedom in the order of applying the Givens rotations, the elements of A can be eliminated by many different orderings. The *independent* (or *disjoint*) rotations induced by a particular ordering can be computed simultaneously provided there are enough processors available. While the number of independent rotations increases with the problem size and changes during the factorization process, the number of processors on a parallel computer is fixed. Therefore, the independent rotations must be statically or dynamically allocated to the processors in some way. The choice of a different ordering and the particular strategy of assigning independent rotations to processors give rise to different parallel algorithms. A number of them have been studied in literature for implementation on hypercube multiprocessors [6,18,71,72].

There are five parallel Givens algorithms proposed in [6,18,71,72]. They are all based on "Givens sequences", which are sequences of Givens rotations where zeros once created are preserved. The two new parallel algorithms proposed in this chapter can be viewed as different implementations of a particular Givens sequence on the hypercube. Both algorithms take full advantage of the hypercube topology and require only *nearest-neighbour* communication. They differ from the algorithms in [6,18,71,72] in communication schemes, data mapping schemes, arithmetic/communication complexities, and work load distribution. In particular, we show how *redundant computation* can be incorporated into the communication algorithm to maintain data proximity so that all processors in the hypercube can simultaneously exchange data with their neighbouring processors. In addition, we show in the second algorithm how the hypercube topology can be used to reduce the computational cost as well as the communication cost of a parallel algorithm.

3.2 Algorithm I

The first algorithm we propose is based on the Givens sequence illustrated in Figure 3.1 by a 16×8 matrix. That is, the nonzero elements below the main diagonal of an m-by-n matrix are eliminated column by column in the order indicated in Figure 3.1, where the (i, k) entries to be zeroed in the k^{th} elimination stage are labelled by the integer k. In the parallel algorithm, the p processors cooperate to annihilate the $m - k$ nonzero elements in column k during the k^{th} elimination stage. If a Givens rotation is applied to the i^{th} row and the j^{th} row to annihilate the leading nonzero $a_{j,k}$ in the j^{th} row, then row i is referred to as the "pivot row".

As usual with parallel algorithms, we would like to achieve a balanced distribution of work load and a low volume of data movement and communication. A uniform work load distribution and a low communication cost contribute directly to the *speed-up*, which is the ultimate goal of a concurrent algorithm.

Since there is no globally shared memory in the hypercube, the data must be distributed among the processing nodes in some way, typically by rows if the computation is row-oriented, or by columns if the computation is column-oriented. *In either case*, there is a decision to be made concerning the way in which the rows or columns are mapped onto the processors. For example, given an m-by-n matrix A and p processors P_0, P_1, P_2, $\cdots$, P_{p-1}, *block-mapping* may be used, where the first m/p rows (or n/p columns) are assigned to processor P_0, the next m/p rows (or n/p columns) are assigned to processor P_1, and so on. Alternatively, *wrap-mapping* may be used, where consecutive rows (or columns) are

$$\begin{pmatrix}
\times & \times & \times & \times & \times & \times & \times & \times \\
1 & \times & \times & \times & \times & \times & \times & \times \\
1 & 2 & \times & \times & \times & \times & \times & \times \\
1 & 2 & 3 & \times & \times & \times & \times & \times \\
1 & 2 & 3 & 4 & \times & \times & \times & \times \\
1 & 2 & 3 & 4 & 5 & \times & \times & \times \\
1 & 2 & 3 & 4 & 5 & 6 & \times & \times \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & \times \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{pmatrix}$$

Figure 3.1: Column-by-column Givens sequence.

assigned to consecutive processors, with assignment "wrapping around" to processor P_0 after a row (or column) is assigned to processor P_{p-1}.

Discussion about various mapping strategies for matrix computations can be found in [11,22,51]. For our purpose, it suffices to observe that if block-mapping is used to assign rows or columns of A to the p processors, then processor P_0 will become idle when the first m/p rows or the first n/p columns of A do not need to be modified any more. The other $p-1$ processors could become idle one after another subsequently for the same reason. On the other hand, the rationale behind the wrap-mapping is to assign the data to the p processors in such a manner that every processor will be doing approximately the same amount of computation and communication throughout the entire process except for the last p steps. Although the p processors become idle one after another in the last p steps, the idle time so incurred is not significant if the work involved for each of these p steps is only a tiny fraction of the total work.

For Algorithm I, we allocate the m rows to the p processors using a wrap-mapping. Although wrap-mapping is not essential for the correctness of the algorithms we propose, it is important for efficiency because the latter depends very much on whether a balanced work load distribution can be maintained throughout the computing process. Figure 3.2 illustrates the wrap mapping of sixteen rows to four processors.

For a given $m \times n$ matrix, Algorithm I has n stages, each stage consisting of nine steps. The nine steps of the k^{th} stage are devised to annihilate the $m-k$ nonzero elements in locations $(k+1,k)$, $(k+2,k)$, ..., and (m,k). Steps 1 to 9 of the k^{th} stage of the algorithm can be divided into two distinct phases, namely an Independent Annihilation Phase (IAP) and a Cooperative Merging Phase (CMP). The IAP corresponds to Step 1, where each processor operates on its assigned data without communicating with other processors in the network. The CMP corresponds to Steps 2 to 9, where it is necessary for each processor to exchange data with its d neighbours if a hypercube of dimension d

$$\begin{pmatrix}
P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 \\
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 \\
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 \\
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 \\
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3
\end{pmatrix}$$

Figure 3.2: Wrap mapping of 16 rows to 4 processors.

is employed. Using a hypercube of dimension $d = 3$, we illustrate in Figure 3.3 the 3 communication steps in the CMP of each stage. We use "$b_{d-1}b_{d-2}\cdots b_0$" to denote the d-bit binary representation of a processor *id*. To accomplish the d exchanges, each processor pairs with another processor whose *id* is different in bit $b_{\ell-1}$, $\ell = d, d-1, \ldots, 1$. For example, processor P_0 accomplishes the 3 exchanges by communicating with processors P_4, P_2 and P_1 sequentially. The processor *id*'s of the latter three processors are 100, 010 and 001 respectively. During each communication step, the $p/2$ pairs of processors exchange data using $p/2$ disjoint channels. In our description of Steps 2 to 9 below, it will become apparent that in the CMP of the k^{th} stage, the data to be exchanged between each pair of processors always consist of $(n-k+1)$ floating-point numbers. It also turns out that the computational work performed by each processor after each exchange of data is the same.

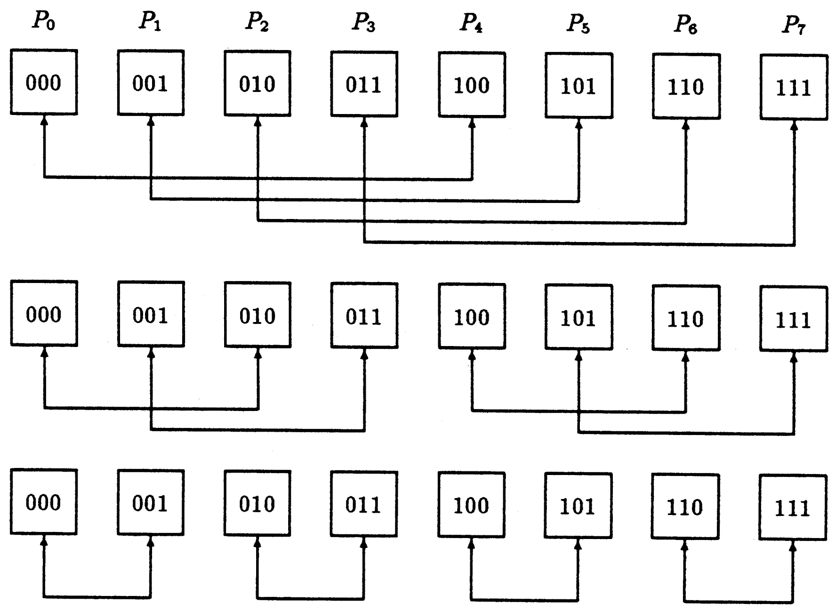


Figure 3.3: The d communication steps in the CMP ($d = 3$).

We now describe the steps for the k^{th} stage. An example is used along the way to help explain each of the nine steps. After the details for the k^{th} stage are given, we shall use an example to demonstrate how the entire algorithm works and summarize the features of the communication algorithm we propose.

Step 1 (IAP) Among all of the rows with row number $i \geq k$, each processor uses the lowest numbered row as the pivot row to eliminate all of the off-diagonal nonzero elements in the k^{th} column of its remaining rows by Givens rotations.

Using the example in Figure 3.2, the action of processor P_0 in the first elimination stage is illustrated in Figure 3.4. The leading nonzeros in rows 5, 9 and 13 are annihilated by applying Givens rotations sequentially to the three pairs of rows, namely {row 1, row 5}, {row 1, row 9} and {row 1, row 13}.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,8} \\ a_{5,1} & a_{5,2} & \cdots & a_{5,8} \\ a_{9,1} & a_{9,2} & \cdots & a_{9,8} \\ a_{13,1} & a_{13,2} & \cdots & a_{13,8} \end{pmatrix} \longrightarrow \begin{pmatrix} \tilde{a}_{1,1} & \tilde{a}_{1,2} & \cdots & \tilde{a}_{1,8} \\ 0 & \tilde{a}_{5,2} & \cdots & \tilde{a}_{5,8} \\ 0 & \tilde{a}_{9,2} & \cdots & \tilde{a}_{9,8} \\ 0 & \tilde{a}_{13,2} & \cdots & \tilde{a}_{13,8} \end{pmatrix}$$

Figure 3.4: The action of P_0 in the first elimination stage.

Therefore, no communication is needed in the IAP. At the end of this step, every processor has one row with a nonzero in the k^{th} column. We refer to this row as the "local pivot row". The (i,k) nonzero entries in these local pivot rows (except for element (k,k)) are to be annihilated at Steps 2 to 9 in the CMP. Figure 3.5 displays the remaining nonzero elements at the end of Step 1 for a 16×8 example when $p = 4$ and $k = 1$. The elements of each local pivot row are marked by its assigned processor P_i, $0 \leq i \leq 3$. The entries $(2,1)$, $(3,1)$ and $(4,1)$ are to be zeroed in the CMP.

$$\begin{pmatrix}
P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 & P_0 \\
P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 & P_1 \\
P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 & P_2 \\
P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 & P_3 \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times
\end{pmatrix}$$

Figure 3.5: The distribution of local pivot rows at the end of Step 1 (IAP).

Step 2 (CMP) $\ell \leftarrow d$, where d is the dimension of the hypercube.

Step 3 (CMP) Every processor sends its current local pivot row to the processor whose *id* differs in bit $b_{\ell-1}$. It also receives a row from the other processor.

For the example in Figure 3.5, when $k = 1$, $\ell = d = 2$, rows 1 and 3 are thereby made available to both P_0 and P_2, and rows 2 and 4 are thereby made available to both P_1 and P_3 at the end of this step. We shall denote this pair of rows as $\tilde{a}_{\rho_1,*}$ and $\tilde{a}_{\rho_2,*}$, where ρ_1 and ρ_2 are their respective row numbers in the matrix A, and $\rho_1 < \rho_2$.

Step 4 (CMP) Each processor computes a Givens rotation to eliminate the element $\tilde{a}_{\rho_2,k}$ by executing the following algorithm.

```
if $|\tilde{a}_{\rho_2,k}| \geq |\tilde{a}_{\rho_1,k}|$ then
    $t \leftarrow |\tilde{a}_{\rho_1,k}|/|\tilde{a}_{\rho_2,k}|$
    $s_{\rho_2,k} \leftarrow 1/\sqrt{1+t^2}$
    $c_{\rho_2,k} \leftarrow s_{\rho_2,k} \times t$
else
    $t \leftarrow |\tilde{a}_{\rho_2,k}|/|\tilde{a}_{\rho_1,k}|$
    $c_{\rho_2,k} \leftarrow 1/\sqrt{1+t^2}$
    $s_{\rho_2,k} \leftarrow c_{\rho_2,k} \times t$
```

If row ρ_2 is originally assigned to this processor, then row ρ_1, row ρ_2, $c_{\rho_2,k}$ and $s_{\rho_2,k}$ are saved so that row ρ_2 can be updated in Step 9.

For the given example, when $k = 1$ and $\ell = d = 2$, P_2 saves row 1, row 3, $c_{3,1}$ and $s_{3,1}$, and P_3 saves row 2, row 4, $c_{4,1}$ and $s_{4,1}$.

Step 5 (CMP) Each processor "updates" row ρ_1 by executing the following algorithm. The updated row ρ_1 becomes the "current" local pivot row.

```
for $j = k, k+1, \ldots, n$ do
    $\tilde{a}_{\rho_1,j} \leftarrow c_{\rho_2,k}\tilde{a}_{\rho_1,j} + s_{\rho_2,k}\tilde{a}_{\rho_2,j}$
```

For the example in Figure 3.5, when $k = 1$ and $\ell = 2$, row 1 is modified to become the current local pivot row in P_0 and P_2, whereas row 2 is modified to become the current local pivot row in P_1 and P_3. Note that redundant computation is performed.

Step 6 (CMP) $\ell \leftarrow \ell - 1$.

Step 7 (CMP) If $\ell > 1$ then go to Step 3.

Step 8 (CMP) Each processor sends its current local pivot row to the processor whose *id* differs in bit b_0.

Step 9 (CMP) The processor which was originally assigned row k executes the algorithms given in Step 4 and 5 to update row k. Each other processor modifies the row originally assigned to it by executing the following algorithm. (Note that this row is either the higher numbered row currently received or the one saved at Step 4.)

```
for $j = k, k+1, \ldots, n$ do
    $\tilde{a}_{\rho_2,j} \leftarrow c_{\rho_2,k}\tilde{a}_{\rho_1,j} - s_{\rho_2,k}\tilde{a}_{\rho_2,j}$
```

For the given example, when $k = 1$ and $\ell = 1$, processors P_0, P_1, P_2 and P_3 will modify respectively rows 1, 2, 3 and 4 simultaneously in this step.

To demonstrate how Steps 2 to 9 in the CMP work in general, we trace the pair of rows (ρ_1, ρ_2) for three elimination stages in Figures 3.6–3.8, where a hypercube of dimension 3 is employed. The rows to be finally updated in Step 9 are quoted in Figures 3.6–3.8. Note that each processor has exactly one quoted row during each elimination stage.

Processors	$\ell = 3$ (ρ_1, ρ_2)	$\ell = 2$ (ρ_1, ρ_2)	$\ell = 1$ (ρ_1, ρ_2)	Step 9 ρ_1 or ρ_2
P_0	(1, 5)	(1, 3)	('1', 2)	'1'
P_1	(2, 6)	(2, 4)	(1 ,'2')	'2'
P_2	(3, 7)	(1,'3')	(1 , 2)	'3'
P_3	(4, 8)	(2,'4')	(1 , 2)	'4'
P_4	(1,'5')	(1, 3)	(1 , 2)	'5'
P_5	(2,'6')	(2, 4)	(1 , 2)	'6'
P_6	(3,'7')	(1, 3)	(1 , 2)	'7'
P_7	(4,'8')	(2, 4)	(1 , 2)	'8'

Figure 3.6: The 1^{st} elimination stage.

Processors	$\ell = 3$ (ρ_1, ρ_2)	$\ell = 2$ (ρ_1, ρ_2)	$\ell = 1$ (ρ_1, ρ_2)	Step 9 ρ_1 or ρ_2
P_0	(5,'9')	(3, 5)	(2 , 3)	'9'
P_1	(2, 6)	(2, 4)	('2', 3)	'2'
P_2	(3, 7)	(3, 5)	(2 ,'3')	'3'
P_3	(4, 8)	(2,'4')	(2 , 3)	'4'
P_4	(5, 9)	(3,'5')	(2 , 3)	'5'
P_5	(2,'6')	(2, 4)	(2 , 3)	'6'
P_6	(3,'7')	(3, 5)	(2 , 3)	'7'
P_7	(4,'8')	(2, 4)	(2 , 3)	'8'

Figure 3.7: The 2^{nd} elimination stage.

Processors	$\ell = 3$ (ρ_1, ρ_2)	$\ell = 2$ (ρ_1, ρ_2)	$\ell = 1$ (ρ_1, ρ_2)	Step 9 ρ_1 or ρ_2
P_0	(5, '9')	(3, 5)	(3 , 4)	'9'
P_1	(6,'10')	(4, 6)	(3 , 4)	'10'
P_2	(3, 7)	(3, 5)	('3', 4)	'3'
P_3	(4, 8)	(4, 6)	(3 ,'4')	'4'
P_4	(5, 9)	(3,'5')	(3 , 4)	'5'
P_5	(6, 10)	(4,'6')	(3 , 4)	'6'
P_6	(3, '7')	(3, 5)	(3 , 4)	'7'
P_7	(4, '8')	(4, 6)	(3 , 4)	'8'

Figure 3.8: The 3^{rd} elimination stage.

We summarize the features of the proposed communication algorithm below.

1. Given a hypercube of dimension d, each processor has d neighbours and the communication algorithm executed at each elimination stage involves exactly d steps. At each of the d steps, each processor in the hypercube exchanges a message of the same length with a neighbouring processor. Since the $p/2$ exchanges at each step involve $p/2$ distinct pairs of processors and use $p/2$ separate communication channels, they can occur *simultaneously*.

2. Each processor communicates with all of its d neighbouring processors in *exactly the same order* at every elimination stage. For example, in Figures 3.6 to 3.8, processor P_0 always communicates with processors P_4, then P_2, and lastly P_1.

3. It is clear from our description of the algorithm and from Figures 3.6 to 3.8 that some rows are *redundantly* updated by more than one processor concurrently. For example, in Figure 3.6, when $\ell = 3$ row 1 is redundantly updated by processor P_4, when $\ell = 2$

row 1 is redundantly updated by processors P_2, P_4 and P_6.

The redundant computation allows us to use the same communication algorithm for every elimination stage regardless of which processor owns the lowest numbered row. For example, in Figure 3.7, among all the rows with row number $i \geq 2$, the second row of A is the lowest numbered row which needs to be modified last and the greatest number of times. Although the lowest numbered row is now located in processor P_1 instead of processor P_0 as in the first elimination stage, the same communication algorithm combined with the redundant computation makes sure that row 2 is properly updated, as are the other rows. In addition, the data each processor needs are always located in a neighbouring processor in the hypercube.

4. The proposed communication algorithm can be easily generalized for Algorithm II which we shall present in section 3.4.

5. When $k > n - p + 1$, the processors run out of rows one by one. Because the algorithm above requires all processors to participate in maintaining data proximity in the remaining stages, a uniform treatment of all cases can be achieved by simply assigning dummy data to processors which would otherwise finish earlier. With this technique the algorithm we described for the k^{th} elimination stage can remain the same for $1 \leq k \leq n$.

It is appropriate to point out that Algorithm I differs from the distributed Givens algorithms proposed by Chamberlain and Powell [6] and Pothen et al. [71,72] in the communication algorithm we employ for merging the local pivot rows. According to the timing results to be discussed in section 3.6, our idea of using otherwise idle processors to perform redundant computation appears to be effective in keeping the communication algorithm simple, efficient and versatile for use in a generalized version of Algorithm I and other algorithms.

3.3 Performance Analysis of Algorithm I

In this section we analyze the performance of Algorithm I in factoring an $m \times n$ matrix using a hypercube of dimension d. Letting p denote the total number of processors, we have $p = 2^d$. For convenience we assume that m and n are integral multiples of p. In our analysis we assume that the time required for a multiplicative floating-point operation is τ, and that the time required to transmit η floating-point numbers from one processor to a neighbouring processor is $\eta\lambda + \beta$, where β is the start-up time for sending a message and λ is the time needed to transmit a floating-point number across one link between adjacent processors in the hypercube.

We shall compare the performance of Algorithm I with the sequential Givens algorithm, for which the total arithmetic cost is given by

$$\begin{aligned} T_s(m,n) &= 4\tau \sum_{k=1}^{n} (m-k)(n-k+1) \\ &= \frac{2n^2(3m-n)}{3}\tau + 2mn\tau - 2n^2\tau - \frac{4}{3}n\tau \,, \end{aligned} \tag{3.1}$$

when $m \geq n$, and

$$\begin{aligned} \tilde{T}_s(m,n) &= 4\tau \sum_{k=1}^{m} (m-k)(n-k+1) \\ &= \frac{2m^2(3n-m)}{3}\tau - 2mn\tau + 2m^2\tau - \frac{4}{3}m\tau \,, \end{aligned} \tag{3.2}$$

when $m \leq n$.

The wrap mapping of rows of the matrix to the p processors dictates that the size of the largest submatrix in an individual processor is $(m/p) \times (n-k+1)$ for the k^{th} elimination stage when $k = 1, 2, \cdots, p$, and $(m/p - 1) \times (n-k+1)$ when $k = p+1, p+2, \cdots, 2p$, and so on. When $m \geq n$, the arithmetic cost of Algorithm I is thus given by

$$T_I^A(m,n,p) = 4\tau \sum_{k=1}^{n/p} \sum_{j=1}^{p} \left(\frac{m}{p} - k\right)(n - p(k-1) - j + 1)$$

$$\begin{aligned} & + 2\tau \sum_{k=1}^{n} d(n-k+1) \\ = & \frac{2n^2(3m-n)}{3p}\tau + n^2 d\tau - n^2\tau + \left(\frac{2mn-n^2}{p}\right)\tau \\ & + O(np)\,, \end{aligned} \tag{3.3}$$

where $d = \log_2 p$. For each of the n elimination stages, d "nearest-neighbour" exchanges are required, involving in the k^{th} stage a row of size $(n-k+1)$. The communication cost is therefore given by

$$\begin{aligned} T_I^C(n,p) &= \beta \sum_{k=1}^{n} 2d + \lambda \sum_{k=1}^{n} 2d(n-k+1) \\ &= 2nd\beta + \left(n^2+n\right) d\lambda. \end{aligned} \tag{3.4}$$

When $m \le n$, we have

$$\begin{aligned} \tilde{T}_I^A(m,n,p) &= 4\tau \sum_{k=1}^{m/p} \sum_{j=1}^{p} \left(\frac{m}{p}-k\right)(n-p(k-1)-j+1) \\ & \quad + 2\tau \sum_{k=1}^{m} d(n-k+1) \\ &= \frac{2m^2(3n-m)}{3p}\tau + \left(2mn-m^2\right) d\tau - \left(2mn-m^2\right)\tau \\ & \quad + \frac{m^2}{p}\tau + O(mp) \end{aligned} \tag{3.5}$$

and

$$\begin{aligned} \tilde{T}_I^C(m,n,p) &= \beta \sum_{k=1}^{m} 2d + \lambda \sum_{k=1}^{m} 2d(n-k+1) \\ &= 2md\beta + \left(2mn-m^2+m\right) d\lambda\,. \end{aligned} \tag{3.6}$$

When $m = n$, we can further simplify equations (3.3), (3.4), (3.5) and (3.6) as

$$\begin{aligned} T_I^A(n,n,p) &= \tilde{T}_I^A(n,n,p) \\ &= \frac{4n^3}{3p}\tau + n^2(\log_2 p)\tau - n^2\tau + \frac{n^2}{p}\tau + O(np) \end{aligned} \tag{3.7}$$

and

$$\begin{aligned} T_I^C(n,p) &= \tilde{T}_I^C(n,n,p) \\ &= 2n\,(\log_2 p)\,\beta + n^2\,(\log_2 p)\,\lambda + n\,(\log_2 p)\,\lambda\,. \end{aligned} \tag{3.8}$$

Each processor does the same amount of arithmetic, performs the same number of sends and receives, and sends and receives the same amount of data using separate communication channels at every step. Thus, the work load balance of this scheme is guaranteed. The wrap mapping of the upper triangular or trapezoidal factor is also maintained.

Comparing the parallel time $\left(T_I^A + T_I^C\right)$ with the optimal time (T_s/p) for the case $m \geq n$, and $\left(\tilde{T}_I^A + \tilde{T}_I^C\right)$ with the optimal time $\left(\tilde{T}_s/p\right)$ for the case $m \leq n$, we conclude that Algorithm I is optimal in its leading term.

3.4 Algorithm II

Algorithm II exploits an embedding of a $\gamma_1 \times \gamma_2$ rectangular grid within the hypercube. The objective of this embedding is to map the processors to a grid so that the processors in each column or each row of the grid form a subcube. Such a mapping allows us to apply the communication algorithm employed in Algorithm I to each subset of processors which form either a row or a column in the embedded grid. If we let $d = d_1 + d_2$, $\gamma_1 = 2^{d_1}$, and $\gamma_2 = 2^{d_2}$, this objective is achieved by requiring that the *id*'s of the processors in the same row differ in only the right-most d_2 bits, and that the *id*'s of the processors in the same column differ in only the left-most d_1 bits. Note that the communication algorithm we shall propose requires only a subcube topology in each column and each row of the grid. Therefore, the adjacent processors in the embedded $\gamma_1 \times \gamma_2$ grid are not required to be neighbours in the hypercube, and the embedding can be done by mapping processors to the grid row by row or column by column following the natural order of their processor *id*. Figure 3.9 displays a 4×4 grid consisting of 16 processors, and Figure 3.10 displays an 8×4 grid consisting of

32 processors.

0000	0001	0010	0011	P_0	P_1	P_2	P_3
0100	0101	0110	0111	P_4	P_5	P_6	P_7
1000	1001	1010	1011	P_8	P_9	P_{10}	P_{11}
1100	1101	1110	1111	P_{12}	P_{13}	P_{14}	P_{15}

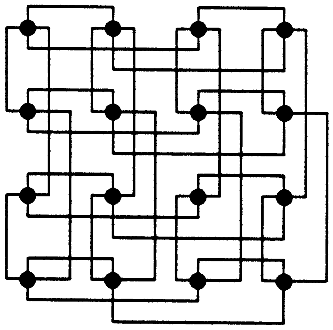


Figure 3.9: The embedding of a 4×4 grid in a hypercube.

From now on, we shall refer to the processor in the (i, j) position of the grid by $P(i, j)$ or by its *id*. We shall use $P(i, *)$ to denote the subcube consisting of the processors assigned to the i^{th} row of the grid, and $P(*, j)$ to denote the subcube consisting of the processors assigned to the j^{th} column of the grid.

Algorithm II is based on the same Givens sequence employed in Algorithm I. In Algorithm II, the data mapping strategy can be understood as wrapping the rows of the $m \times n$ matrix around the γ_1 subcubes, namely $P(1, *)$, $P(2, *)$, $\cdots$, and $P(\gamma_1, *)$. Within each subcube $P(i, *)$, the elements of each allocated row are wrapped around the γ_2 processors according to their column subscripts. Figure 3.11 illustrates this data mapping strategy for a 16×16 matrix on a 4×4 processor grid.

00000	00001	00010	00011	P_0	P_1	P_2	P_3
00100	00101	00110	00111	P_4	P_5	P_6	P_7
01000	01001	01010	01011	P_8	P_9	P_{10}	P_{11}
01100	01101	01110	01111	P_{12}	P_{13}	P_{14}	P_{15}
10000	10001	10010	10011	P_{16}	P_{17}	P_{18}	P_{19}
10100	10101	10110	10111	P_{20}	P_{21}	P_{22}	P_{23}
11000	11001	11010	11011	P_{24}	P_{25}	P_{26}	P_{27}
11100	11101	11110	11111	P_{28}	P_{29}	P_{30}	P_{31}

Figure 3.10: The embedding of an 8 × 4 grid in a hypercube.

$$\begin{pmatrix}
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 \\
P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} \\
P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} \\
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 \\
P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} \\
P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} \\
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 \\
P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} \\
P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} \\
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 \\
P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} \\
P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15}
\end{pmatrix}$$

Figure 3.11: The wrap mapping of a 16 × 16 matrix to a 4 × 4 processor grid.

Algorithm II has n stages, each consisting of two phases. In the Independent Annihilation Phase (IAP) of the k^{th} elimination stage, each subcube $P(i,*)$ independently annihilates the nonzero elements below the main diagonal in the k^{th} column using its lowest numbered row as the local pivot row. The algorithm for the IAP requires no communication between the processors in different subcubes $P(i_1,*)$ and $P(i_2,*)$, where $i_1 \neq i_2$, whereas the processors consisting of each subcube $P(i,*)$ need to communicate among themselves during the IAP. We assume in the example of this section that a 4×4 processor grid is embedded in a hypercube. Figure 3.12 displays the data assigned to the subcube $P(1,*)$, which is a submatrix consisting of rows 1, 5, 9 and 13 of a 16×16 matrix A. Within the subcube, the columns of the assigned submatrix are wrapped around the γ_2 processors of $P(1,*)$. Letting $a_{i,j}$ denote the (i,j) element of matrix A, Figures 3.13–3.16 display the data assigned to processors $P(1,1)$, $P(1,2)$, $P(1,3)$ and $P(1,4)$ respectively.

$$\left(\begin{array}{cccccccccccccccc}
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times
\end{array}\right)$$

Figure 3.12: The wrap mapping of the submatrix within subcube $P(1,*)$.

$$\begin{pmatrix} a_{1,1} & a_{1,5} & a_{1,9} & a_{1,13} \\ a_{5,1} & a_{5,5} & a_{5,9} & a_{5,13} \\ a_{9,1} & a_{9,5} & a_{9,9} & a_{9,13} \\ a_{13,1} & a_{13,5} & a_{13,9} & a_{13,13} \end{pmatrix}$$

Figure 3.13: Data assigned to processor P_0.

$$\begin{pmatrix} a_{1,2} & a_{1,6} & a_{1,10} & a_{1,14} \\ a_{5,2} & a_{5,6} & a_{5,10} & a_{5,14} \\ a_{9,2} & a_{9,6} & a_{9,10} & a_{9,14} \\ a_{13,2} & a_{13,6} & a_{13,10} & a_{13,14} \end{pmatrix}$$

Figure 3.14: Data assigned to processor P_1.

$$\begin{pmatrix} a_{1,3} & a_{1,7} & a_{1,11} & a_{1,15} \\ a_{5,3} & a_{5,7} & a_{5,11} & a_{5,15} \\ a_{9,3} & a_{9,7} & a_{9,11} & a_{9,15} \\ a_{13,3} & a_{13,7} & a_{13,11} & a_{13,15} \end{pmatrix}$$

Figure 3.15: Data assigned to processor P_2.

$$\begin{pmatrix} a_{1,4} & a_{1,8} & a_{1,12} & a_{1,16} \\ a_{5,4} & a_{5,8} & a_{5,12} & a_{5,16} \\ a_{9,4} & a_{9,8} & a_{9,12} & a_{9,16} \\ a_{13,4} & a_{13,8} & a_{13,12} & a_{13,16} \end{pmatrix}$$

Figure 3.16: Data assigned to processor P_3.

To eliminate the nonzeros $a_{5,1}, a_{9,1}$, and $a_{13,1}$, processor P_0 computes the multiplier pairs $\{c_{\rho,1}, s_{\rho,1}\}$ for $\rho = 5, 9, 13$ and updates the "local pivot element", $a_{1,1}$, by the following algorithm.

```
**if** $|a_{\rho,1}| \geq |a_{1,1}|$ **then**
    $t \leftarrow |a_{1,1}|/|a_{\rho,1}|$
    $s_{\rho,1} \leftarrow 1/\sqrt{1+t^2}$
    $c_{\rho,1} \leftarrow s_{\rho,1} \times t$
    $a_{1,1} \leftarrow |a_{1,1}|\sqrt{1+t^2}$
**else**
    $t \leftarrow |a_{\rho,1}|/|a_{1,1}|$
    $c_{\rho,1} \leftarrow 1/\sqrt{1+t^2}$
    $s_{\rho,1} \leftarrow c_{\rho,1} \times t$
    $a_{1,1} \leftarrow |a_{1,1}|\sqrt{1+t^2}$
```

A message containing the "updated" $a_{1,1}$ and *all of the computed multiplier pairs* is then made available to every processor in the subcube $P(1,*)$ using the following recursive exchange algorithm. Recall that the *id*'s of processors in each subcube $P(i,*)$ differ only in their right-most d_2 bits. Suppose every processor has a message which must be made available to every other processor in the subcube, the basic recursive exchange algorithm works as follows.

```
$\ell \leftarrow d_2$
while $\ell > 0$ do
    send (my message) to processor with id different
        from my id in bit $b_{\ell-1}$.
    receive a message
    $\ell \leftarrow \ell - 1$
```

To broadcast the multiplier pairs and the local pivot element to all processors within each subcube $P(i, *)$, we modify the basic recursive exchange algorithm in the following manner. The processor which has the pivot column composes a message consisting of the computed multipliers and the local pivot element, whereas the other processors simply prepare a dummy message. The modified algorithm works as follows.

```
$\ell \leftarrow d_2$
while $\ell > 0$ do
    send (my message) to processor with id different
        from my id in bit $b_{\ell-1}$.
    receive a message
    if (my message) is (dummy message)
        (my message) $\leftarrow$ (received message)
    $\ell \leftarrow \ell - 1$
```

The subcubes $P(1, *)$, $P(2, *)$, $P(3, *)$ and $P(4, *)$ each perform essentially the same IAP task with respect to their own data independently and simultaneously. The communication

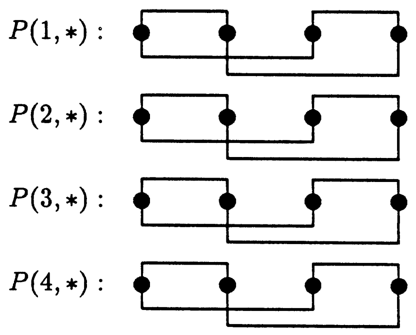


Figure 3.17: The communication channels employed by the subcubes during the IAP.

channels employed by the four subcubes are displayed in Figure 3.17. Therefore, after d_2 exchanges within each subcube, all processors have the multipliers they need to update the remaining elements independently. The resulting matrix has at most γ_1 rows with a nonzero in the k^{th} column. For $k = 1$, Figure 3.18 illustrates the mapping of the γ_1 local pivot rows at the end of the IAP.

$$\begin{pmatrix}
P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\
P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 \\
P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} \\
P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times
\end{pmatrix}$$

Figure 3.18: End of the IAP during the first elimination stage.

The remaining off-diagonal nonzero elements in the first column can now be eliminated by the γ_2 subcubes $P(*,j)$, $j = 1,2,\cdots,\gamma_2$, independently and simultaneously. Figure 3.19 displays the data to be affected in the subcube $P(*,1) = \{P_0, P_4, P_8, P_{12}\}$.

$$\begin{pmatrix}
P_0 & \times & \times & \times & P_0 & \times & \times & \times & P_0 & \times & \times & \times & P_0 & \times & \times & \times \\
P_4 & \times & \times & \times & P_4 & \times & \times & \times & P_4 & \times & \times & \times & P_4 & \times & \times & \times \\
P_8 & \times & \times & \times & P_8 & \times & \times & \times & P_8 & \times & \times & \times & P_8 & \times & \times & \times \\
P_{12} & \times & \times & \times & P_{12} & \times & \times & \times & P_{12} & \times & \times & \times & P_{12} & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times
\end{pmatrix}$$

Figure 3.19: Data distribution in subcube $P(*,1)$.

Comparing Figure 3.19 with Figure 3.5, clearly the task to be performed by the subcube $P(*,1)$ is essentially Steps 2 through 9 of Algorithm I. Of course, the data now refers to the submatrix in Figure 3.20, and the subcube $P(*,1)$ is of dimension $d/2$, and the *id*'s of the processors in this subcube differ only in their *left-most* $d/2$ bits. It is straightforward

$$\begin{pmatrix}
a_{1,1} & a_{1,5} & a_{1,9} & a_{1,13} \\
a_{2,1} & a_{2,5} & a_{2,9} & a_{2,13} \\
a_{3,1} & a_{3,5} & a_{3,9} & a_{3,13} \\
a_{4,1} & a_{4,5} & a_{4,9} & a_{4,13}
\end{pmatrix}$$

Figure 3.20: Data to be processed by subcube $P(*,1)$.

to modify Step 2 through 9 of Algorithm I to reflect these differences. Recall that during the IAP, each updated local pivot element was sent to all processors in the respective subcube together with the multipliers. When $k = 1$, $\{a_{1,1}, a_{2,1}, a_{3,1}, a_{4,1}\}$ are thus available in subcube $P(*,j)$ for all j. Figure 3.21 displays the data distribution in the subcube $P(*,2)$. Each processor in the subcube $P(*,2)$ has one more element in addition to the data originally assigned.

$$\begin{pmatrix}
P_1 & P_1 & \times & \times & \times & P_1 & \times & \times & \times & P_1 & \times & \times & \times & P_1 & \times & \times \\
P_5 & P_5 & \times & \times & \times & P_5 & \times & \times & \times & P_5 & \times & \times & \times & P_5 & \times & \times \\
P_9 & P_9 & \times & \times & \times & P_9 & \times & \times & \times & P_9 & \times & \times & \times & P_9 & \times & \times \\
P_{13} & P_{13} & \times & \times & \times & P_{13} & \times & \times & \times & P_{13} & \times & \times & \times & P_{13} & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times
\end{pmatrix}$$

Figure 3.21: Data distribution in subcube $P(*,2)$.

Comparing Figure 3.21 with Figure 3.5, observe that the task to be performed by the subcube $P(*,2)$ is again essentially the same as described in Steps 2 through 9 of Algorithm I. The data now refers to the matrix in Figure 3.22. The elements $\{a_{1,1}, a_{2,1}, a_{3,1}, a_{4,1}\}$ are needed in Step 4 to compute the multipliers. Therefore, the strategy of sending the "updated" local pivot elements together with the multiplers in the IAP is the most economic way to make these elements available to the respective processors. Similar arguments apply to the subcubes $P(*,3)$ and $P(*,4)$. The communication channels employed by the four subcubes during the CMP are displayed in Figure 3.23.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,6} & a_{1,10} & a_{1,14} \\ a_{2,1} & a_{2,2} & a_{2,6} & a_{2,10} & a_{2,14} \\ a_{3,1} & a_{3,2} & a_{3,6} & a_{3,10} & a_{3,14} \\ a_{4,1} & a_{4,2} & a_{4,6} & a_{4,10} & a_{4,14} \end{pmatrix}$$

Figure 3.22: Data to be processed by subcube $P(*,2)$.

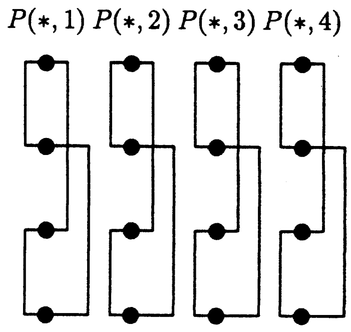


Figure 3.23: The communication channels employed by the subcubes during the CMP.

Figure 3.24 displays the remaining elements of A after $n - \gamma_2$ elimination stages, where $p = 16$ and $\gamma_2 = 4$ in this example. Since $(\gamma_1 + \gamma_2 - 2i - 1)$ more processors will become idle at the end of each of the last γ_2 elimination stages, where $0 \leq i \leq (\gamma_2 - 1)$, the idle time is proportional to the amount of work each processor is assigned for one stage. When $m = n$ and $\gamma_1 = \gamma_2$, each processor has exactly one element to be zeroed or modified in the last γ_2 elimination stages. The idle time thus remains constant regardless of the size of the matrix. When $m > n$, γ_1 processors would become idle after each of the last γ_2 elimination stages. Because the elements remaining in each processor after $n - \gamma_2$ stages is proportional to $(m-n)/\gamma_1$, the idle time grows linearly with m for fixed γ_1 if $(m-n) \gg n$. When $m < n$, because γ_2 processors would become idle following each of the last γ_1 stages, the idle time grows linearly with n for fixed γ_2 if $(n - m) \gg m$. As we shall see in the next section, γ_1 and γ_2 are to be chosen according to the shape of the matrix so that the performance of Algorithm II is optimized. The possibility of choosing γ_1 and γ_2 in proportion to m and n implies that the linear growth rate represents the worst case. Note that in the actual implementation of Algorithm II, the communication algorithms in both IAP and CMP phases require all processors to participate regardless of whether there is any computational work remaining for a particular processor. The idle time we mentioned above refers to the duration of time from the moment a processor has completely processed the assigned data of matrix A to the moment the parallel program ends. Thus the time such a processor spends working on dummy data is considered as idle time.

3.5 Performance Analysis of Algorithm II

In this section we analyze the performance of Algorithm II in factoring an $m \times n$ matrix on a $\gamma_1 \times \gamma_2$ grid embedded in a hypercube of dimension d, where $d = d_1 + d_2$, $\gamma_1 = 2^{d_1}$ and $\gamma_2 = 2^{d_2}$. Letting p denote the total number of processors, we have $p = \gamma_1\gamma_2 = 2^d$. As before, we assume that m and n are integral multiples of p. The definitions for τ, β and λ

$$\left(\begin{array}{cccccccccccccccc}
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & & & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & & & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & & & & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & & & & & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & & & & & & \times & \times & \times & \times & \times & \times \\
 & & & & & & & & & & & \times & \times & \times & \times & \times \\
 & & & & & & & & & & & & P_0 & P_1 & P_2 & P_3 \\
 & & & & & & & & & & & & P_4 & P_5 & P_6 & P_7 \\
 & & & & & & & & & & & & P_8 & P_9 & P_{10} & P_{11} \\
 & & & & & & & & & & & & P_{12} & P_{13} & P_{14} & P_{15}
\end{array}\right)$$

Figure 3.24: Data distribution for the last four elimination stages.

are as given in section 3.3.

3.5.1 The case $m \geq n$

When $m \geq n$, we first consider the case $\gamma_1 \geq \gamma_2$. To analyze the total arithmetic cost of Algorithm II, let us consider the k^{th} elimination stage. During the IAP phase, the γ_1 subcubes $P(1,*)$, $P(2,*)$, $\cdots$, $P(\gamma_1,*)$ are performing essentially the same task on the γ_1 different submatrices independently and simultaneously. Within each subcube $P(i,*)$ the submatrix is further divided among the γ_2 processors consisting of the subcube. Letting $(\gamma_1/\gamma_2) = \alpha$, the wrap mapping of rows and columns of the matrix to the processor grid dictates that the size of the largest submatrix in an individual processor is $(m/\gamma_1) \times (n/\gamma_2)$ for $k = 1, 2, \cdots, \gamma_2$, $(m/\gamma_1) \times (n/\gamma_2 - 1)$ for $k = \gamma_2 + 1, \gamma_2 + 2, \cdots, 2\gamma_2$, $\cdots$, and $(m/\gamma_1 - 1) \times (n/\gamma_2 - \alpha)$ for $k = \gamma_1 + 1, \gamma_1 + 2, \cdots, \gamma_1 + \gamma_2$, and so on. The total arithmetic cost for the IAP is thus

$$T_{IAP}^{A}(m,n,\gamma_1,\gamma_2) \quad = \quad 4\tau \sum_{k=1}^{n/\gamma_1} \sum_{j=1}^{\alpha} \sum_{i=1}^{\gamma_2} \left(\frac{m}{\gamma_1} - k\right) \left(\frac{n}{\gamma_2} - \alpha(k-1) - j + 1\right)$$

$$\begin{aligned}
&= \frac{2n^2(3m-n)}{3p}\tau + \frac{2mn}{\gamma_1}\tau - n^2\left(\frac{\gamma_1+\gamma_2}{\gamma_1\gamma_2}\right)\tau \\
&\quad - n\tau - \frac{n}{3}\left(\frac{\gamma_1}{\gamma_2}\right)\tau \\
&= \frac{2n^2(3m-n)}{3p}\tau + \frac{2mn}{\gamma_1}\tau - n^2\left(\frac{\gamma_1+\gamma_2}{p}\right)\tau \\
&\quad + O(n)\,. \qquad (3.9)
\end{aligned}$$

Recall that the multiplier pairs together with the updated local pivot element must be made available to the γ_2 processors within each subcube $P(i,*)$ using the recursive exchange algorithm. The total communication cost for the IAP is thus given by

$$\begin{aligned}
T^C_{IAP}(m,n,\gamma_1,\gamma_2) &= \beta\sum_{k=1}^{n} 2d_2 + \lambda\sum_{k=1}^{n/\gamma_1}\sum_{j=1}^{\gamma_1} 2d_2\left(\frac{m}{\gamma_1} - k + 1\right) \\
&= 2nd_2\beta + \left(\frac{2mn-n^2}{\gamma_1} + n\right) d_2\lambda\,. \qquad (3.10)
\end{aligned}$$

In the cooperative merging phase (CMP) following the IAP, the γ_1 processors in each subcube $P(*,j)$, $1 \le j \le \gamma_2$, perform essentially Steps 2 through 9 of Algorithm I. When $k = 1$, every processor applies a Givens rotation d_1 times to a row of size at most (n/γ_2+1), and exchanges a row of the same size with a neighbouring processor d_1 times. The longest row in an individual processor is $(n/\gamma_2 + 1)$ for $k = 1, 2, \cdots, \gamma_2$, and (n/γ_2) for $k = \gamma_2 + 1, \gamma_2 + 2, \cdots, 2\gamma_2$, and so on. We thus have

$$\begin{aligned}
T^A_{CMP}(n,\gamma_1,\gamma_2) &= 2\tau\sum_{k=1}^{n/\gamma_2}\sum_{j=1}^{\gamma_2} d_1\left(\frac{n}{\gamma_2} - k + 2\right) \\
&= \left(\frac{n^2}{\gamma_2} + 3n\right) d_1\tau \qquad (3.11)
\end{aligned}$$

and

$$\begin{aligned}
T^C_{CMP}(n,\gamma_1,\gamma_2) &= \beta\sum_{k=1}^{n} 2d_1 + \lambda\sum_{k=1}^{n/\gamma_2}\sum_{j=1}^{\gamma_2} 2d_1\left(\frac{n}{\gamma_2} - k + 2\right) \\
&= 2nd_1\beta + \left(\frac{n^2}{\gamma_2} + 3n\right) d_1\lambda\,. \qquad (3.12)
\end{aligned}$$

The total arithmetic and communication costs for Algorithm II are thus given by

$$\begin{aligned} T_{II}^{A}(m,n,\gamma_1,\gamma_2) &= T_{IAP}^{A}(m,n,\gamma_1,\gamma_2) + T_{CMP}^{A}(n,\gamma_1,\gamma_2) \\ &= \frac{2n^2(3m-n)}{3p}\tau + \frac{n^2}{\gamma_2}d_1\tau + \frac{2mn}{\gamma_1}\tau - n^2\left(\frac{\gamma_1+\gamma_2}{p}\right)\tau \\ &\quad + O(nd_1) \end{aligned} \tag{3.13}$$

and

$$\begin{aligned} T_{II}^{C}(m,n,\gamma_1,\gamma_2) &= T_{IAP}^{C}(m,n,\gamma_1,\gamma_2) + T_{CMP}^{C}(n,\gamma_1,\gamma_2) \\ &= 2nd\beta + \frac{2mn-n^2}{\gamma_1}d_2\lambda + \frac{n^2}{\gamma_2}d_1\lambda + O(nd)\,. \end{aligned} \tag{3.14}$$

Since $\gamma_1\gamma_2 = p$, the parallel time of Algorithm II can be expressed as a function of m, n, γ_1 and p as given in equation (3.15).

$$\begin{aligned} T_{II}(m,n,\gamma_1,p) &= T_{II}^{A}(m,n,\gamma_1,\gamma_2) + T_{II}^{C}(m,n,\gamma_1,\gamma_2) \\ &= \frac{2n^2(3m-n)}{3p}\tau + \frac{2mn-n^2}{\gamma_1}d_2\lambda + \frac{n^2}{\gamma_2}d_1(\tau+\lambda) \\ &\quad + \frac{2mn}{\gamma_1}\tau - n^2\left(\frac{\gamma_1+\gamma_2}{p}\right)\tau + O(nd) \\ &= \frac{2n^2(3m-n)}{3p}\tau + \frac{2mn-n^2}{\gamma_1}(d-d_1)\lambda + \frac{n^2}{p}\gamma_1 d_1(\tau+\lambda) \\ &\quad + \frac{2mn}{\gamma_1}\tau - n^2\left(\frac{{\gamma_1}^2+p}{p\gamma_1}\right)\tau + O(n\log_2 p)\,. \end{aligned} \tag{3.15}$$

If $\gamma_1 \le \gamma_2$, T_{IAP}^{C}, T_{CMP}^{A} and T_{CMP}^{C} remain the same as given by equations (3.10), (3.11) and (3.12), whereas T_{IAP}^{A} is now computed by equation (3.16), where we use $\tilde{\alpha}$ to denote γ_2/γ_1.

$$\begin{aligned} T_{IAP}^{A}(m,n,\gamma_1,\gamma_2) &= 4\tau\sum_{k=1}^{n/\gamma_2}\sum_{j=1}^{\tilde{\alpha}}\sum_{i=1}^{\gamma_1}\left(\frac{m}{\gamma_1} - \tilde{\alpha}(k-1) - j\right)\left(\frac{n}{\gamma_2} - k + 1\right) \\ &= \frac{2n^2(3m-n)}{3p}\tau + \frac{2mn}{\gamma_1}\tau - n^2\left(\frac{\gamma_1+\gamma_2}{\gamma_1\gamma_2}\right)\tau \\ &\quad - n\tau - \frac{n}{3}\left(\frac{\gamma_2}{\gamma_1}\right)\tau\,. \end{aligned} \tag{3.16}$$

Comparing (3.16) with (3.9), we see that they differ only in one of the low order terms. Therefore, for $m \geq n$, we shall use (3.15) to compute $T_{II}(m, n, \gamma_1, p)$ for all values of γ_1. Our analysis of the communication cost, as summarized by equation (3.14), indicates that the total number of messages exchanged between each pair of processors is independent of the choice of γ_1. Accordingly, the contribution of *start-up* time to the communication cost of Algorithm II remains the same for all values of γ_1.

One objective of our analysis is to find the value of γ_1 which minimizes the cost of the parallel algorithm. To find γ_1, we set

$$\frac{\partial T_{II}(m, n, \gamma_1, p)}{\partial \gamma_1} = 0$$

and obtain

$$\begin{aligned} f\left(p, \gamma_1, \frac{m}{n}, \frac{\tau}{\lambda}\right) &= a\left(\frac{\tau}{\lambda}+1\right)(\ln \gamma_1 + 1){\gamma_1}^2 - \left(\frac{\tau}{\lambda}\right){\gamma_1}^2 + ap\left(2\frac{m}{n}-1\right)\ln \gamma_1 \\ &\quad - p\left(\frac{\tau}{\lambda} + \log_2 p + a\right)\left(2\frac{m}{n}-1\right) \\ &= 0, \end{aligned} \tag{3.17}$$

where $a = 1/(\ln 2)$.

Therefore, for fixed p, $\frac{m}{n}$ and $\frac{\tau}{\lambda}$, the value of γ_1 which minimizes $T_{II}(m, n, \gamma_1, p)$ can be obtained by finding the solution to $f(p, \gamma_1, \frac{m}{n}, \frac{\tau}{\lambda}) = 0$, which is a nonlinear equation in the variable γ_1. Since γ_1 must be an integral power of 2, we choose $\gamma_1 = 2^k$ for $0 \leq k \leq \log_2 p$, with 2^k as close as possible to the solution of $f(p, \gamma_1, \frac{m}{n}, \frac{\tau}{\lambda}) = 0$. Although the "optimal" γ_1 chosen in this manner does not necessarily minimize $T_{II}(m, n, \gamma_1, p)$, the numerical experiments to be presented in Section 7 indicate that for each test problem, the execution time of Algorithm II using the chosen γ_1 either achieves or is very close to the actual minimum over all possible values of γ_1.

In order to see how the optimal γ_1 varies with the ratio τ/λ, we list in Table 3.1 the numerical solution to $f(p, \gamma_1, \frac{m}{n}, \frac{\tau}{\lambda}) = 0$ for different values of τ/λ when p and m/n remain fixed. The optimal γ_1's chosen based on these numerical solutions are displayed in Table 3.2.

From Tables 3.1 and 3.2 we observe that the optimal γ_1 appears to be very insensitive to the ratio τ/λ.

Numerical solution to $f\left(p, \gamma_1, \frac{m}{n}, \frac{\tau}{\lambda}\right) = 0$								
$m \geq n$		Different values of τ/λ						
p	m/n	1000	10	5	1	0.2	0.1	0.0
16	1.0	2.86	3.06	3.20	3.62	3.89	3.94	4.0
16	1.5	3.71	3.95	4.11	4.63	4.97	5.03	5.11
16	19.8	12.33	12.58	12.76	13.35	13.76	13.83	13.92
64	1.0	4.86	5.42	5.80	6.95	7.70	7.84	8.0
64	1.5	6.41	7.11	7.58	9.04	10.01	10.19	10.39
64	19.8	22.41	23.79	24.78	27.96	30.13	30.54	31.00
128	1.0	6.42	7.30	7.89	9.68	10.85	11.06	11.30
128	1.5	8.53	9.64	10.38	12.67	14.19	14.47	14.79
128	19.8	30.38	32.87	34.62	40.24	44.08	44.81	45.63
256	1.0	8.53	9.89	10.78	13.50	15.29	15.62	18.54
256	1.5	11.41	13.13	14.27	17.78	20.11	20.55	21.04
256	19.8	41.30	45.52	48.44	57.77	64.16	65.37	66.74
1024	1.0	15.34	18.41	20.40	26.41	30.40	31.15	32.00
1024	1.5	20.69	24.64	27.23	35.10	40.35	41.35	42.47
1024	19.8	76.92	87.78	95.15	118.41	134.34	137.39	140.82

Table 3.1: Numerical solution to $f\left(p, \gamma_1, \frac{m}{n}, \frac{\tau}{\lambda}\right) = 0$.

3.5.2 The case $m \leq n$

Similarly, when $m \leq n$, we have the cases $\gamma_1 \geq \gamma_2$ and $\gamma_1 \leq \gamma_2$. Although our derivation below is for the case $\gamma_1 \leq \gamma_2$, $\tilde{T}_{IAP}^{A}$ is different only in one of the low order terms when

Predicted Optimal γ_1								
$m \geq n$		Different values of τ/λ						
p	m/n	1000	10	5	1	0.2	0.1	0
16	1.0	2	4	4	4	4	4	4
16	1.5	4	4	4	4	4	4	4
16	19.8	16	16	16	16	16	16	16
64	1.0	4	4	4	8	8	8	8
64	1.5	8	8	8	8	8	8	8
64	19.8	16	16	32	32	32	32	32
128	1.0	8	8	8	8	8	8	8
128	1.5	8	8	8	16	16	16	16
128	19.8	32	32	32	32	32	32	32
256	1.0	8	8	8	16	16	16	16
256	1.5	8	16	16	16	16	16	16
256	19.8	32	32	64	64	64	64	64
1024	1.0	16	16	16	32	32	32	32
1024	1.5	16	16	32	32	32	32	32
1024	19.8	64	64	64	128	128	128	128

Table 3.2: Predicted optimal γ_1 when $m \geq n$.

$\gamma_1 \geq \gamma_2$ and $\tilde{T}^C_{IAP}$, $\tilde{T}^A_{CMP}$ and $\tilde{T}^C_{CMP}$ remain unchanged. We shall therefore use the following formula for all values of γ_1. Letting $(\gamma_2/\gamma_1) = \tilde{\alpha}$, we have

$$\begin{aligned}
\tilde{T}^A_{IAP}(m,n,\gamma_1,\gamma_2) &= 4\tau \sum_{k=1}^{m/\gamma_2} \sum_{j=1}^{\tilde{\alpha}} \sum_{i=1}^{\gamma_1} \left(\frac{m}{\gamma_1} - \tilde{\alpha}(k-1) - j\right)\left(\frac{n}{\gamma_2} - k + 1\right) \\
&= \frac{2m^2(3n-m)}{3p}\tau + \frac{m^2}{\gamma_1}\tau + \frac{m^2 - 2mn}{\gamma_2}\tau \\
&\quad + O(m)\,, \qquad (3.18) \\
\tilde{T}^C_{IAP}(m,n,\gamma_1,\gamma_2) &= \beta \sum_{k=1}^{m} 2d_2 + \lambda \sum_{k=1}^{m/\gamma_1} \sum_{j=1}^{\gamma_1} 2d_2 \left(\frac{m}{\gamma_1} - k + 1\right) \\
&= 2md_2\beta + \left(\frac{m^2}{\gamma_1} + m\right) d_2\lambda\,, \qquad (3.19) \\
\tilde{T}^A_{CMP}(m,n,\gamma_1,\gamma_2) &= 2\tau \sum_{k=1}^{m/\gamma_2} \sum_{j=1}^{\gamma_2} d_1 \left(\frac{n}{\gamma_2} - k + 2\right) \\
&= \left(\frac{2mn - m^2}{\gamma_2} + 3m\right) d_1\tau\,, \qquad (3.20)
\end{aligned}$$

and

$$\begin{aligned}
\tilde{T}^C_{CMP}(m,n,\gamma_1,\gamma_2) &= \beta \sum_{k=1}^{m} 2d_1 + \lambda \sum_{k=1}^{m/\gamma_2} \sum_{j=1}^{\gamma_2} 2d_1 \left(\frac{n}{\gamma_2} - k + 2\right) \\
&= 2md_1\beta + \left(\frac{2mn - m^2}{\gamma_2} + 3m\right) d_1\lambda\,. \qquad (3.21)
\end{aligned}$$

The total parallel arithmetic cost and communication cost are thus given by

$$\begin{aligned}
\tilde{T}^A_{II}(m,n,\gamma_1,\gamma_2) &= \tilde{T}^A_{IAP}(m,n,\gamma_1,\gamma_2) + \tilde{T}^A_{CMP}(m,n,\gamma_1,\gamma_2) \\
&= \frac{2m^2(3n-m)}{3p}\tau + \left(\frac{2mn - m^2}{\gamma_2}\right) d_1\tau + \frac{m^2}{\gamma_1}\tau \\
&\quad + \left(\frac{m^2 - 2mn}{\gamma_2}\right)\tau + O(md_1) \qquad (3.22)
\end{aligned}$$

and

$$\tilde{T}^C_{II}(m,n,\gamma_1,\gamma_2) = \tilde{T}^C_{IAP}(m,n,\gamma_1,\gamma_2) + \tilde{T}^C_{CMP}(m,n,\gamma_1,\gamma_2)$$

$$
\begin{aligned}
&= 2nd\beta + \left(\frac{2mn - m^2}{\gamma_2}\right) d_1\lambda + \frac{m^2}{\gamma_1} d_2\lambda \\
&\quad + O\left(m \log_2 p\right) . \qquad (3.23)
\end{aligned}
$$

When $m \leq n$, the parallel time of Algorithm II can again be expressed as a function of m, n, γ_1, and p as given in equation (3.24).

$$
\begin{aligned}
\tilde{T}_{II}\left(m,n,\gamma_1,p\right) &= \tilde{T}_{II}^{A}\left(m,n,\gamma_1,\gamma_2\right) + \tilde{T}_{II}^{C}\left(m,n,\gamma_1,\gamma_2\right) \\
&= \frac{2m^2(3n-m)}{3p}\tau + \left(\frac{2mn - m^2}{\gamma_2}\right) d_1(\tau + \lambda) + \frac{m^2}{\gamma_1}(\tau + d_2\lambda) \\
&\quad + \left(\frac{m^2 - 2mn}{\gamma_2}\right)\tau + O\left(m \log_2 p\right) \\
&= \frac{2m^2(3n-m)}{3p}\tau + \left(\frac{2mn - m^2}{p}\right) d_1\gamma_1(\tau + \lambda) \\
&\quad + \frac{m^2}{\gamma_1}(\tau + (d - d_1)\lambda) + \left(\frac{m^2 - 2mn}{p}\right)\gamma_1\tau \\
&\quad + O\left(m \log_2 p\right) . \qquad (3.24)
\end{aligned}
$$

The value of γ_1 which minimizes $\tilde{T}_{II}\left(m,n,\gamma_1,p\right)$ can now be found by setting

$$
\frac{\partial \tilde{T}_{II}(m,n,\gamma_1,p)}{\partial \gamma_1} = 0\,,
$$

from which we obtain

$$
\begin{aligned}
\tilde{f}\left(p,\gamma_1,\frac{n}{m},\frac{\tau}{\lambda}\right) &= a\left(\frac{\tau}{\lambda}+1\right)\left(2\frac{n}{m}-1\right)(\ln\gamma_1 + 1){\gamma_1}^2 - \frac{\tau}{\lambda}\left(2\frac{n}{m}-1\right){\gamma_1}^2 \\
&\quad + ap(\ln\gamma_1) - p\left(\frac{\tau}{\lambda} + \log_2 p + a\right) \\
&= 0\,. \qquad (3.25)
\end{aligned}
$$

Similarly, for fixed p, $\frac{n}{m}$ and $\frac{\tau}{\lambda}$, the value of γ_1 which minimizes $\tilde{T}_{II}(m,n,\gamma_1,p)$ can be obtained by finding the solution to $\tilde{f}\left(p,\gamma_1,\frac{n}{m},\frac{\tau}{\lambda}\right) = 0$. As before, γ_1 must be an integral power of 2, and we choose it as close to the solution of equation (3.25) as possible.

In order to see how the optimal γ_1 varies with the relative speeds of computation and communication, we list in Table 3.3 the numerical solution to $\tilde{f}\,(p, \gamma_1, \frac{n}{m}, \frac{\tau}{\lambda}) = 0$ for different values of τ/λ when p and n/m remain fixed. The optimal γ_1's chosen based on these numerical solutions are displayed in Table 3.4. Again, the optimal γ_1 is quite insensitive to the ratios of τ to λ.

Numerical solution to $\tilde{f}\,(p, \gamma_1, \frac{n}{m}, \frac{\tau}{\lambda}) = 0$								
$m \leq n$		Different values of τ/λ						
p	n/m	1000	10	5	1	0.2	0.1	0
16	1.0	2.86	3.06	3.20	3.62	3.89	3.94	4.0
16	1.5	2.24	2.40	2.51	2.84	3.05	3.09	3.13
16	19.8	0.98	1.02	1.04	1.10	1.14	1.14	1.15
64	1.0	4.86	5.42	5.80	6.95	7.70	7.84	8.0
64	1.5	3.71	4.16	4.46	5.35	5.93	6.04	6.16
64	19.8	1.37	1.51	1.59	1.85	2.01	2.03	2.06
128	1.0	6.42	7.30	7.89	9.68	10.85	11.07	11.31
128	1.5	4.86	5.56	6.02	7.40	8.30	8.47	8.65
128	19.8	1.68	1.90	2.04	2.46	2.71	2.75	2.81
256	1.0	8.53	9.89	10.78	13.50	15.29	15.62	16.0
256	1.5	6.42	7.48	8.18	10.26	11.63	11.88	12.17
256	19.8	2.10	2.45	2.67	3.30	3.69	3.76	3.84
1024	1.0	15.34	18.41	20.40	26.41	30.40	31.15	32.0
1024	1.5	11.42	13.79	15.32	19.90	22.91	23.47	24.11
1024	19.8	3.46	4.23	4.70	6.07	6.93	7.09	7.27

Table 3.3: Numerical solution to $\tilde{f}\,(p, \gamma_1, \frac{n}{m}, \frac{\tau}{\lambda}) = 0$.

Predicted optimal γ_1								
$m \leq n$		Different values of τ/λ						
p	n/m	1000	10	5	1	0.2	0.1	0
16	1.0	2	4	4	4	4	4	4
16	1.5	2	2	2	2	4	4	4
16	19.8	1	1	1	1	1	1	1
64	1.0	4	4	4	8	8	8	8
64	1.5	4	4	4	4	4	8	8
64	19.8	1	2	2	2	2	2	2
128	1.0	8	8	8	8	8	8	8
128	1.5	8	8	8	8	8	8	8
128	19.8	2	2	2	2	2	2	2
256	1.0	8	8	8	16	16	16	16
256	1.5	8	8	8	8	8	8	16
256	19.8	2	2	2	4	4	4	4
1024	1.0	16	16	16	32	32	32	32
1024	1.5	8	16	16	16	16	16	32
1024	19.8	4	4	4	8	8	8	8

Table 3.4: Predicted optimal γ_1 when $m \leq n$.

Comparing the leading term of $T_{II}(m, n, \gamma_1, p)$ with the leading term of $T_s(m, n)/p$ for the case $m \geq n$, and from comparing the leading term of $\tilde{T}_{II}(m, n, \gamma_1, p)$ with the leading term of of $\tilde{T}_s(m, n)/p$ for the case $m \leq n$, it can be concluded that Algorithm II is optimal in its leading term.

3.5.3 Analysis of Storage Requirements

According to our data mapping strategy for Algorithm II, the rows and columns of a given matrix are wrap-mapped to γ_1 and γ_2 processors respectively. Therefore, the processors run out of rows and/or columns one by one in the last γ_1 or γ_2 elimination stages. As explained earlier, our communication algorithm requires all processors to participate in maintaining data proximity in every stage. We thus adopted the strategy of assigning one more row of all zeros and one more column of all zeros to each processor. The largest submatrix assigned to a processor is therefore $\left\lceil \frac{m}{\gamma_1} + 1 \right\rceil$ by $\left\lceil \frac{n}{\gamma_2} + 1 \right\rceil$. In addition to storing the submatrix, each processor also needs buffer space for sending and receiving the multipliers (in the IAP) and sending, receiving and saving the pivot row (in the CMP). There are also integer overhead incurred by choosing particular data structures which facilitate a clean implementation. Such overhead amounts to $(2m/\gamma_1 + n/\gamma_2 + O(p))$ more integers in our implementation. In the analysis below we consider the total storage requirement on a single processor as the sum of the primary storage for data and the overhead storage for buffers, the extra zero row and column, and the integer overhead. The low order terms which neither vary with m nor vary with n are ignored. The total storage is thus a function of m, n, γ_1 and γ_2. Since $\gamma_2 = p/\gamma_1$, for a given $m \times n$ matrix it is desirable to find the value of γ_1 that minimizes the total storage. We assume that the space for storing an integer is the same as the space for storing a floating-point number in the following analysis.

Lemma 3.1 *For any given m, n, and* $p = \gamma_1 \times \gamma_2$, *the total storage requirement of Algorithm II on each node processor is given by*

$$\frac{mn}{p} + 7\frac{m}{\gamma_1} + 4\frac{n}{\gamma_2} + 7\,, if\; 2\left(\frac{m}{\gamma_1} + 1\right) \geq \frac{n}{\gamma_2} + 1\,,$$

and

$$\frac{mn}{p} + 3\frac{m}{\gamma_1} + 6\frac{n}{\gamma_2} + 5\,, if\; 2\left(\frac{m}{\gamma_1} + 1\right) \leq \frac{n}{\gamma_2} + 1\,.$$

Proof: As noted earlier, each processor is assigned a submatrix of size $(m/\gamma_1 + 1) \times (n/\gamma_2 + 1)$. The buffer space for sending and receiving the multipliers is twice the size of the largest set of multipliers, i.e., $2 \times 2\,(m/\gamma_1 + 1)$. Similarly, the buffer space for sending and receiving the pivot row is twice the row length of each submatrix, i.e., $2 \times (n/\gamma_2 + 1)$. Since the buffer space for multipliers can be re-used for sending and receiving pivot rows, it is sufficient to have enough storage for the larger one of these two buffers. In addition to the buffer space for sending and receiving the pivot row, in the CMP we need extra buffer space of $2 \times (n/\gamma_2 + 1)$ floating-point numbers to save the pair of rows in case the updating is delayed. Summing up the data storage, buffer storage and the integer overhead given above we obtain the results in the lemma. □

Theorem 3.2 *For any given m, n and p, the storage requirement of Algorithm II is minimized by* $\gamma_1 = 2^k$, *where* $k \in [0, \log_2 p]$ *is chosen so that* γ_1 *is as close as possible to* $\sqrt{\frac{7mp}{6n}}$.

Proof: Assuming that the buffer space for the multipliers and the pivot row cannot be overlapped, we seek γ_1 to minimize

$$S(m, n, \gamma_1, \gamma_2) = \frac{mn}{p} + 7\frac{m}{\gamma_1} + 6\frac{n}{\gamma_2} + 7\,.$$

Substituting $\gamma_2 = p/\gamma_1$, and setting

$$\frac{\partial S(m, n, \gamma_1, \gamma_2)}{\partial \gamma_1} = 0\,,$$

we obtain

$$\gamma_1 = \sqrt{\frac{7mp}{6n}}\,.$$

□

Recall that the data of the coefficient matrix only require storage for mn/p floating-point numbers per processor. Thus it is necessary to address the question of whether the overhead storage is a significant fraction of the primary storage for the chosen γ_1. In Corollary 3.3, we give the formula for computing the ratio of the overhead storage to the primary storage mn/p when $\gamma_1 = \sqrt{7mp/(6n)}$.

Corollary 3.3 *When $\gamma_1 = \sqrt{\frac{7mp}{6n}}$, the ratio of the overhead storage to the primary storage is given by*

$$10.8\sqrt{\frac{p}{mn}} + 7\frac{p}{mn}\,,\ \textit{if}\ 2\left(\frac{m}{\gamma_1}+1\right) \geq \frac{n}{\gamma_2}+1\,,$$

and

$$9.3\sqrt{\frac{p}{mn}} + 5\frac{p}{mn}\,,\ \textit{if}\ 2\left(\frac{m}{\gamma_1}+1\right) \leq \frac{n}{\gamma_2}+1\,.$$

Proof: Substituting γ_1 in Lemma 3.1 by $\sqrt{7mp/(6n)}$ and γ_2 by p/γ_1, we obtain

$$S(m,n,\gamma_1,p) = \frac{mn}{p} + 10.8\sqrt{\frac{mn}{p}} + 7\,,$$

and

$$\tilde{S}(m,n,\gamma_1,p) = \frac{mn}{p} + 9.3\sqrt{\frac{mn}{p}} + 5\,.$$

The results in the corollary are obtained by computing

$$\frac{S(m,n,\gamma_1,p) - (mn/p)}{mn/p}$$

and

$$\frac{\tilde{S}(m,n,\gamma_1,p) - (mn/p)}{mn/p}\,.$$

□

Since γ_1 is unlikely to be exactly equal to $\sqrt{7mp/(6n)}$ in practice, we computed the actual overhead storage and compared with the results obtained from the formula given by Corollary 3.3. Letting γ_1^s denote the γ_1 chosen by Theorem 3.2, we list in Table 3.5 the values of $\sqrt{7mp/(6n)}$, γ_1^s and the predicted and actual ratio of overhead storage to primary storage for a set of matrices. The two ratios given as the predicted percentages are obtained by substituting $\gamma_1 = \sqrt{7mp/(6n)}$ in each of the two formulas given in Corollary 3.3. The actual percentage given in the last column is computed by substituting the chosen γ_1^s into the appropriate formula in Lemma 3.1.

p	m	n	$\sqrt{7mp/(6n)}$	predicted percentage	γ_1^s	actual percentage
16	500	500	4.32	7.5%–8.7%	4	8.8%
16	600	400	5.29	7.6%–8.9%	4	9.7%
16	400	600	3.53	7.6%–8.9%	4	8.7%
64	1000	1000	8.64	7.5%–8.7%	8	8.8%
64	1200	800	10.57	7.6%–8.9%	8	9.7%
64	800	1200	7.05	7.6%–8.9%	8	8.7%
64	1980	100	38.44	16.9%–19.6%	32	20.7%
64	100	1980	1.93	16.9%–19.6%	2	19.5%

Table 3.5: Predicted and actual overhead storage.

For easy comparison, we give in Table 3.6 the value of γ_1^s as well as the predicted optimal γ_1 for the set of matrices listed in Table 3.5.

Corollary 3.3 implies that the overhead storage will be insignificant if m, n are large and $p \ll min\{m, n\}$.

				Different values of τ/λ						
				1000	10	5	1	0.2	0.1	0
p	m	n	γ_1^s	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1
16	500	500	4	2	4	4	4	4	4	4
16	600	400	4	4	4	4	4	4	4	4
16	400	600	4	2	2	2	2	4	4	4
64	1000	1000	8	4	4	4	8	8	8	8
64	1200	800	8	8	8	8	8	8	8	8
64	800	1200	8	4	4	4	4	4	8	8
64	1980	100	32	16	16	32	32	32	32	32
64	100	1980	2	1	2	2	2	2	2	2

Table 3.6: Predicted γ_1^s and predicted optimal γ_1.

3.6 Numerical Experiments

Our experiments were performed on a 64-processor Intel iPSC Hypercube. The node processors employ Intel 80286/80287 chip sets which run at 8 MHz. Each node has 512K bytes of memory. The Intel hypercube is presently a single-user subsystem. Algorithm II was implemented in FORTRAN. Algorithm I is a special case of Algorithm II when the two-dimensional processor grid is chosen to be of dimension p by 1. The programs were compiled using the Ryan-McFarland FORTRAN compiler. We provide timing results for single-precision and double-precision implementations. The maximum run time over all the node processors is reported as the parallel execution time for each test problem. The factorization time reported does not include the time for initialization and data generation.

The execution times (in seconds) of the serial and parallel algorithms are denoted by T_s and T respectively, and as in previous sections, m and n denote the number of rows and columns of each test matrix and γ_1 and γ_2 denote the number of processors along each dimension of the two-dimensional grid embedded in the hypercube.

Our experiments were designed to measure speed-up, and demonstrate how the aspect ratio of the processor grid affects the performance of Algorithm II. We show that when the predicted optimal aspect ratios are used, the execution time and the storage requirement either coincide with or are very close to the actual minimum as the theory developed in previous sections predicts.

3.6.1 The Measurement of Serial Time

Table 3.7 reports the serial time T_s for each randomly generated test matrix. When $\gamma_1 = \gamma_2 = 1$, the two-dimensional processor grid degenerates to a single processor, and Algorithm II involves only the independent annihilation phase (IAP) on a single processor. Since inter-processor communication is not needed during the IAP, the code for the IAP running on one node indeed implements the sequential Givens algorithm. We thus measure T_s by the execution time of the parallel code running on a 1×1 grid.

Due to the limited memory on a single node, the largest matrix we could factor using one processor was about 200 by 200 in single precision or 150 by 150 in double precision. In order to measure the speed-up and efficiency of the parallel algorithm, we needed to estimate the serial factorization time of much larger matrices. For any square matrix of dimension n, we approximated the factorization time using the formulae

$$T_s(n) \approx c_1 n^3 + c_2 n^2 + c_3 n + c_4 \,, \tag{3.26}$$

where c_1, c_2, c_3 and c_4 were obtained in the following manner. First note that by equating $T_s(n)$ to the known execution times for $n = 100, 125, 150, 175$ and 200 (for single-precision implementation) or $n = 50, 75, 100, 125$ and 150 (for double-precision implementation), we obtain five equations and four unknowns. By finding the least-squares solution to the overdetermined system of equations we obtain the coefficients $\{c_1, c_2, c_3, c_4\}$. The estimated $T_s(n)$ are compared with the actual execution times in Table 3.8. Since the node processors

The Sequential Givens Algorithm					
Single Precision			Double Precision		
m	n	T_s (sec)	m	n	T_s (sec)
100	100	67.500	50	50	10.800
125	125	130.465	75	75	35.400
150	150	223.890	100	100	82.600
175	175	353.760	125	125	160.000
200	200	526.095	150	150	274.800
90	60	26.830	60	40	10.100
120	80	62.010	90	60	32.640
135	90	87.550	120	80	75.700
160	120	174.600	135	90	107.020
240	160	477.510	160	120	213.905
60	90	25.300	40	60	9.360
80	120	59.300	60	90	30.995
90	135	84.200	80	120	72.800
120	160	170.600	90	135	103.310
160	240	467.000	120	160	209.510

Table 3.7: Execution times of the sequential Givens algorithm.

on the hypercube do not support multiprogramming, the execution times measured on a node are consistent and reproducible. This feature allows us to obtain accurate estimates based on a relatively small set of samples.

The Sequential Givens Algorithm							
Single Precision				Double Precision			
m	n	T_s (sec)	Estimated T_s	m	n	T_s (sec)	Estimated T_s
100	100	67.500	67.500 sec	50	50	10.800	10.806 sec
125	125	130.465	130.467 sec	75	75	35.400	35.377 sec
150	150	223.890	223.887 sec	100	100	82.600	82.634 sec
175	175	353.760	353.762 sec	125	125	160.000	159.977 sec
200	200	526.095	526.095 sec	150	150	274.800	274.806 sec

Table 3.8: Measured and estimated times of the sequential Givens algorithm.

3.6.2 The Effect of the Aspect Ratio of the Processor Grid

In this section we present numerical experiments to demonstrate the effect on the execution time of Algorithm II induced by varying the aspect ratio of the processor grid. Table 3.9 gives the timing results obtained from the single-precision implementation of Algorithm II. Table 3.10 gives the double-precision timing results. The minimum execution time for each test matrix is marked by an asterisk (*).

Recall that for given m, n and p, the predicted optimal γ_1 varies for different values of τ/λ. In Table 3.2 and Table 3.4 we computed the predicted values of the optimal γ_1 for the ratios of τ/λ ranging from 0 ($\tau \ll \lambda$) to 1000 ($\tau \gg \lambda$). For easy comparison with the actual optimal execution time T^*, we let γ_1^ℓ denote the smallest predicted optimal γ_1, γ_1^u denote the largest predicted optimal γ_1, and label the execution time corresponding to γ_1^ℓ or γ_1^u as T_ℓ or T_u respectively for each test matrix in Tables 3.9 and 3.10.

Some timing results are missing in the tables. In some cases, we did not obtain the execution time because of storage limitation. In particular, the maximum number of bytes that may be sent in a single message on the hypercube is 16K bytes (4000 single-precision or 2000 double-precision floating-point numbers) and this limit was exceeded for some choices of γ_1 and γ_2. In other cases, for the very large test problems whose factorization is very expensive, we only provide the timing result for the optimal choice of $\gamma_1 \times \gamma_2$ because the effect of the shape of the grid on speed-up and efficiency has been well demonstrated on smaller problems.

Since it may be equally important to minimize the storage requirement on each node processor, it is desirable that γ_1^s in Theorem 3.2 coincides with the choice of γ_1 which minimizes the execution time. In order to see how Algorithm II performs in this aspect, we label the execution time corresponding to γ_1^s as $T_\dagger$ for for each test matrix in Tables 3.9 and 3.10.

It is interesting to see that $T_\dagger$, T_ℓ or T_u either coincide with or are very close to the actual optimal T^* for all test matrices in Tables 3.9 and 3.10. It is also worth noting that by embedding an appropriate processor grid we not only minimize the storage usage and communication/computation cost of the parallel algorithm, but also help balance the work load and reduce processor idle time. The 1980×100 and 100×1980 matrices are examples to demonstrate how a proper choice of γ_1 can reduce the processor idle time. Clearly the choice of a 1×64 grid for the 1980×100 matrix is equivalent to wrapping the 100 columns around the 64 processors where each processor is assigned one column or two columns. In contrast, the choice of a 64×1 grid for the same matrix will assign 30 or 31 rows to each processor. In the former case, because only 36 processors are assigned two columns, starting from the 37^{th} elimination stage, the 64 processors will become idle one by one after each following elimination stage. In the latter case, since only one row from the 30 rows or two rows from the 31 rows could be the pivot rows, each processor has 29 to 31 rows of data to process at each of the 100 elimination stages. The reduction of idle time is thus quite significant while using a 64 × 1 grid for this example. A similar argument applies to the 100 × 1980 example.

Algorithm II								
Single-precision Execution Times (sec), $\gamma_1 \times \gamma_2 = 64$								
m	n	64 × 1	32 × 2	16 × 4	8 × 8	4 × 16	2 × 32	1 × 64
1000	1000	1493	1250	1155	$1146^*_{\dagger,u}$	1185_ℓ	1304	1579
1700	1700				$5258^*_{\dagger,u}$			
1200	800	1224	1075	1021*	$1041_{\dagger,u,\ell}$	1103	1264	1628
800	1200	1493	1178	1051	$1007^*_{\dagger,u}$	1028_ℓ	1096	1270
1980	100	43.8*	$44.8_{\dagger,u}$	48.7_ℓ	60.3	85.4	138.9	251.6
100	1980	196.9	102.4	62.7	46.7	41	$39.4^*_{\dagger,u}$	40.1_ℓ

Table 3.9: Single-precision execution times of Algorithm II.

Tables 3.11 and 3.12 report the estimated speed-up and efficiency for a set of test matrices. The speed-up and efficiency are each computed using

$$\textit{speed-up} = \frac{\textit{Estimated}\, T_s}{T},$$

and

$$\textit{efficiency} = \frac{\textit{speed-up}}{p},$$

where p is the total number of processors employed, and $p = \gamma_1 \times \gamma_2$. Observe in Table 3.12 the 19% decrease in efficiency when a 1000 × 1000 matrix is factored on a 64 × 1 grid compared to factoring a 500 × 500 matrix on a 16 × 1 grid. This is in contrast to the 3% decrease in efficiency when the 16 processors and 64 processors are employed as a 4 × 4 grid and an 8 × 8 grid in factoring the same pair of matrices. The observation above indicates that in this case the choice of the optimal aspect ratio also helps the parallel algorithm to scale to a large number of processors.

Algorithm II						
Double-precision Execution Times (sec), $\gamma_1 \times \gamma_2 = 16$						
m	n	16×1	8×2	4×4	2×8	1×16
500	500	729	680	$676.5^{*}_{\dagger,u}$	702_{ℓ}	779
600	400	631	605^{*}	$615_{\dagger,u,\ell}$	656	757
400	600	686	617	$599^{*}_{\dagger,u}$	610_{ℓ}	657

Double-precision Execution Times (sec), $\gamma_1 \times \gamma_2 = 64$								
m	n	64×1	32×2	16×4	8×8	4×16	2×32	1×64
1000	1000	1858	1549	1423	$1402^{*}_{\dagger,u}$	1448_{ℓ}	1586	1918
1200	1200				$2357^{*}_{\dagger,u}$			
1200	800	1517	1326	1256^{*}	$1270_{\dagger,u,\ell}$			
800	1200	1861		1302	$1238^{*}_{\dagger,u}$	1259_{ℓ}		
1980	100	52.1*	$52.6_{\dagger,u}$	57.6_{ℓ}	71.0	101.5	164.4	
100	1980	249.6	129.8	78.4	57.9	49.9	$47.0^{*}_{\dagger,u}$	48.1_{ℓ}

Table 3.10: Double-precision execution times of Algorithm II.

The Estimated Speed-up and Efficiency of Algorithm II							
Single Precision							
$p = 64$, $m = n = 1000$ $T_s \approx 64,377$ sec = 17 hr 52 min 57 sec							
$\gamma_1 \times \gamma_2$	64×1	32×2	16×4	8×8	4×16	2×32	1×64
T (sec)	1493	1250	1155	1146*	1185	1304	1579
speed-up	43.1	51.5	55.7	56.2*	54.3	49.4	40.8
efficiency	67%	81%	87%	88%*	85%	77%	64%
$p = 64$, $m = n = 1700$ $T_s \approx 315,578$ sec = 3 days 15 hr 39 min 38 sec							
$\gamma_1 \times \gamma_2$	64×1	32×2	16×4	8×8	4×16	2×32	1×64
T (sec)				5258*			
speed-up				60.0*			
efficiency				94%*			

Table 3.11: Estimated speed-up and efficiency of Algorithm II.

The Estimated Speed-up and Efficiency of Algorithm II							
Double Precision							
$p = 16$, $m = n = 500$ $T_s \approx 9,962$ sec = 2 hr 46 min 2 sec							
$\gamma_1 \times \gamma_2$	16 x 1		8 x 2	4 x 4	2 x 8	1 x 16	
T (sec)	729		680	676.5*	702	779	
speed-up	13.7		14.7	14.7*	14.2	12.8	
efficiency	86%		92%	92%*	89%	80%	
$p = 64$, $m = n = 1000$ $T_s \approx 79,318$ sec = 22 hr 1 min 58 sec							
$\gamma_1 \times \gamma_2$	64 x 1	32 x 2	16 x 4	8 x 8	4 x 16	2 x 32	1 x 64
T (sec)	1858	1549	1423	1402*	1448	1586	1918
speed-up	42.7	51.2	55.7	56.6*	54.8	50.0	41.3
efficiency	67%	80%	87%	89%*	86%	78%	65%
$p = 64$, $m = n = 1200$ $T_s \approx 136,953$ sec = 1 day 14 hr 2 min 33 sec							
$\gamma_1 \times \gamma_2$	64 x 1	32 x 2	16 x 4	8 x 8	4 x 16	2 x 32	1 x 64
T (sec)				2357*			
speed-up				58.1*			
efficiency				91%*			

Table 3.12: Estimated speed-up and efficiency of Algorithm II.

3.6.3 Further Enhancement

In [71] Pothen and Raghavan proposed a hybrid algorithm for performing orthogonal decomposition of a rectangular matrix on local-memory multiprocessors. The hybrid algorithm proposed in [71] can be viewed as a variant of Algorithm I. The difference lies in the following two aspects. In the IAP the hybrid scheme uses Householder transformations instead of Givens rotations to reduce the arithmetic cost. In the CMP the hybrid scheme used a different communication scheme in merging the local pivot rows. Since Algorithm I is a special case of Algorithm II when a p-by-1 grid is embedded in the hypercube, the strategy of applying Householder transformations in the IAP can be used to reduce the arithmetic cost of Algorithm II regardless of the choice of γ_1. Furthermore, when $\gamma_2 > 1$, the use of Householder transformations during the IAP can also reduce the communication cost of Algorithm II because there are only half as many multipliers to be communicated within each subcube. In terms of message length, each message to be sent and received during the IAP is reduced by a factor of 2 when Householder transformations are used.

As far as our implementation of Algorithm II is concerned, the code for the IAP involves one single subroutine implementing Givens rotations. Therefore, an enhanced version of Algorithm II is immediately obtained by recoding this subroutine using Householder transformations. Our communication algorithms and the entire CMP of Algorithm II remain unchanged. In this section we report timing results of the enhanced Algorithm II and compare its performance with other schemes.

When $\gamma_1 = \gamma_2 = 1$, the enhanced Algorithm II involves only the IAP phase on one node and thus implements the sequential Householder algorithm. The serial time T_s based on Householder algorithm is therefore measured by the execution time of the parallel code running on a 1x1 grid.

Table 3.13 reports the execution times T_s of the sequential Householder algorithm for some randomly generated test matrices. We again estimated the serial factorization time

T_s for large n-by-n matrices by choosing the coefficients for a cubic polynomial $T_s(n)$ as explained earlier in this section. We compare the estimated $T_s(n)$ with the actual execution times in Table 3.14.

The Sequential Householder Algorithm					
Single Precision			Double Precision		
m	n	T_s (sec)	m	n	T_s (sec)
100	100	60.1	50	50	9.2
125	125	115.3	75	75	29.3
150	150	196.9	100	100	67.5
175	175	310.1	125	125	129.6
200	200	460.0	150	150	221.4
90	60	23.9	60	40	8.5
120	80	54.7	90	60	26.8
135	90	77.0	120	80	61.4
160	120	153,0	135	90	86.5
240	160	415.6	160	120	172.0
60	90	23.2	40	60	8.1
80	120	53.5	60	90	26.1
90	135	75.5	80	120	60.1
120	160	151.3	90	135	84.8
160	240	411.0	120	160	170.1

Table 3.13: Execution times of the sequential Householder algorithm.

In Table 3.15 and 3.16 we show that the aspect ratio of the processor grid has a similar effect on the enhanced Algorithm II. An analysis similar to the one in Section 3.5 can be done in order to obtain reliable estimates for the best γ_1 to use in conjunction with the

The Sequential Householder Algorithm							
Single Precision				Double Precision			
m	n	T_s (sec)	Estimated T_s	m	n	T_s (sec)	Estimated T_s
100	100	60.1	60.1 sec	50	50	9.2	9.2 sec
125	125	115.3	115.3 sec	75	75	29.3	29.3 sec
150	150	196.9	196.9 sec	100	100	67.5	67.5 sec
175	175	310.1	310.1 sec	125	125	129.6	129.6 sec
200	200	460.0	460.0 sec	150	150	221.4	221.4 sec

Table 3.14: Measured and estimated times of the sequential Householder algorithm.

enhanced version of algorithm II. Tables 3.17 and 3.18 report the "estimated" speed-up and efficiency for a set of test matrices.

The Enhanced Algorithm II								
Single-precision Execution Times (sec), $\gamma_1 \times \gamma_2 = 64$								
m	n	64×1	32×2	16×4	8×8	4×16	2×32	1×64
1000	1000	1557	1215	1060	1011*	1017	1084	1257
1700	1700				4580*			
1200	800	1231	1012	922	905*	930	1030	1265
800	1200	1618	1183	986	907	891*	924	1030
1980	100	41.7	39.8*	42.0	48.5	64.4	99.6	175.0
100	1980	221.5	125.1	75.7	50.8	41.0	36.6	36.1*

Table 3.15: Single-precision execution times of the enhanced Algorithm II.

The Enhanced Algorithm II								
Double-precision Execution Times (sec), $\gamma_1 \times \gamma_2 = 16$								
m	n	16×1		8×2	4×4	2×8	1×16	
500	500	653		572	542*	551	596	
600	400	549		500	488*	509	571	
400	600	639		534	488	485*	508	
Double-precision Execution Times (sec), $\gamma_1 \times \gamma_2 = 64$								
m	n	64×1	32×2	16×4	8×8	4×16	2×32	1×64
1000	1000	1819	1399	1214	1138*	1148	1224	1424
1200	1200				1905*			
1200	800	1429		1049	1021*	1051		
800	1200	1915			1031	1006*	1043	
1980	100	46.9	43.7*	45.7	54.5	72.0	113.4	
100	1980	274.5	152.8	89.3	60.6	45.8	40.8	40.4*

Table 3.16: Double-precision execution times of the enhanced Algorithm II.

The Estimated Speed-up and Efficiency of The Enhanced Algorithm II							
Single Precision							
$p = 64$, $m = n = 1000$ $T_s \approx 55,464$ sec = 15 hr 24 min 24 sec							
$\gamma_1 \times \gamma_2$	64 x 1	32 x 2	16 x 4	8 x 8	4 x 16	2 x 32	1 x 64
T (sec)	1557	1215	1060	1011*	1017	1084	1257
speed-up	35.6	45.7	52.3	54.9*	54.5	51.2	44.1
efficiency	56%	71%	82%	86%*	85%	80%	69%
$p = 64$, $m = n = 1700$ $T_s \approx 271,428$ sec = 3 days 3 hr 23 min 48 sec							
$\gamma_1 \times \gamma_2$	64 x 1	32 x 2	16 x 4	8 x 8	4 x 16	2 x 32	1 x 64
T (sec)				4580*			
speed-up				59.3*			
efficiency				93%*			

Table 3.17: Estimated speed-up and efficiency of Algorithm II.

The Estimated Speed-up and Efficiency of The Enhanced Algorithm II							
Double Precision							
$p = 16, m = n = 500$; $T_s \approx 7,873$ sec = 2 hr 11 min 13 sec							
$\gamma_1 \times \gamma_2$	16×1	8×2	4×4	2×8	1×16		
T (sec)	653	572	542*	551	596		
speed-up	12.1	13.8	14.5*	14.3	13.2		
efficiency	76%	86%	91%*	89%	83%		
$p = 64, m = n = 1000$; $T_s \approx 62,426$ sec = 17 hr 20 min 26 sec							
$\gamma_1 \times \gamma_2$	64×1	32×2	16×4	8×8	4×16	2×32	1×64
T (sec)	1819	1399	1214	1138*	1148	1224	1424
speed-up	34.3	44.6	51.4	54.9*	54.4	51.0	43.8
efficiency	54%	70%	80%	86%*	85%	80%	67%
$p = 64, m = n = 1200$; $T_s \approx 107,711$ sec = 1 day 5 hr 55 min 11 sec							
$\gamma_1 \times \gamma_2$	64×1	32×2	16×4	8×8	4×16	2×32	1×64
T (sec)				1905*			
speed-up				56.5*			
efficiency				88%*			

Table 3.18: Estimated speed-up and efficiency of the enhanced Algorithm II.

We next compare the performance of Algorithm II and the enhanced Algorithm II in Tables 3.19 and 3.20. The enhanced version of Algorithm I can be viewed as a FORTRAN implementation (with a different communication scheme) of the hybrid algorithm proposed in [71]. In [71] Pothen and Raghavan implemented the hybrid algorithm in the *C* language and compared its performance with four other schemes including one based on the greedy Givens sequence. The latter can be viewed as a variant of Algorithm I with a different communication scheme.

The timing results listed in Table 3.19 indicate that the enhanced Algorithm II coupled with the optimal choice of γ_1 has the fastest execution time. The possible improvement in execution time by the hybrid scheme over Algorithm I can be seen by comparing the data in column 1 with the data in column 2. Note that when $m/p \ll n$ (e.g. $p = 64$, and $m \times n = 800 \times 1200$ or $m \times n = 100 \times 1980$), the hybrid scheme could become less efficient. The factor contributing to the longer execution time of the hybrid scheme is that in this case each submatrix to be reduced by Householder transformations has dimension $(m/p) \times n$ and when $(m/p) \ll n$, the saving by Householder transformations is relatively small and is less than the different overhead caused by employing Householder transformations instead of Givens rotations. This is not likely to happen when $\gamma_1 \times \gamma_2$ is chosen according to the shape of the matrix as demonstrated by the results for the Enhanced Algorithm II shown in the same Table. The possible improvement by the enhanced Algorithm II over Algorithm II can be seen by comparing the data in column 3 with the data in column 4. As noted earlier, when $\gamma_2 = 1$, the hybrid scheme has lower arithmetic cost but the same communication cost compared to Algorithm I; when $\gamma_2 > 1$, the enhanced Algorithm II not only has lower arithmetic cost but also has lower communication cost compared to Algorithm II. This observation is supported by the timing results in Table 3.19.

In Table 3.20 we list the storage requirement for each of the four schemes. The storage requirement of the enhanced Algorithm II is either the minimum or different from the minimum for less than 0.1%.

Finally, in view of the improvement in execution time and storage requirement by employing Householder transformations in the Independent Annihilation Phase of Algorithm II, the saving by reducing the length of each message in the IAP by a factor of 2 appears to be quite significant. Thus, instead of employing Householder transformations in the IAP, we might reduce the execution time and storage requirement of Algorithm II by simply storing the multiplier pair corresponding to each Givens rotation as a single real number using the economical storage technique proposed by Stewart in [84]. At the cost of compressing and retrieving the rotations, the parallel algorithm employing Givens rotations would have the same communication cost and storage requirement as the one employing Householder transformations in the IAP phase, and their performances would be comparable.

Single-Precision Execution Times (sec)						
p	m	n	$\gamma_1 \times \gamma_2 = p \times 1$		$\gamma_1 \times \gamma_2 =$ optimal choice	
			Algorithm I	Hybrid	Algorithm II	Enhanced II
64	1000	1000	1493	1557	1146	1011*
64	1700	1700			5258	4580*
64	1200	800	1224	1231	1021	905*
64	1980	100	43.8	41.7	43.8	39.8*
64	800	1200	1493	1618	1007	891*
64	100	1980	196.9	221.5	39.4	36.1*
Double-Precision Execution Times (sec)						
p	m	n	$\gamma_1 \times \gamma_2 = p \times 1$		$\gamma_1 \times \gamma_2 =$ optimal choice	
			Algorithm I	Hybrid	Algorithm II	Enhanced II
16	500	500	729	653	676.5	542*
64	1000	1000	1858	1819	1402	1138*
64	1200	1200			2357	1905*
16	600	400	631	549	605	488*
64	1200	800	1517	1429	1256	1021*
64	1980	100	52.1	46.9	52.1	43.7*
16	400	600	686	639	599	485*
64	800	1200	1861	1915	1238	1006*
64	100	1980	249.6	274.5	47.0	40.4*

Table 3.19: Comparing the enhanced Algorithm II with other schemes.

Storage Requirement (in 4-byte words)						
Single-Precision Implementation						
p	m	n	$\gamma_1 \times \gamma_2 = p \times 1$		$\gamma_1 \times \gamma_2 =$ optimal choice	
			Algorithm I	Hybrid	Algorithm II	Enhanced II
64	1000	1000	22606	22606	17808	17546*
64	1700	1700	56750	56750	48258	47822*
64	1200	800	20618	20618	17246	16946*
64	1980	100	4366	4366	4366	4212*
64	800	1200	23394	23394	17108*	17121
64	100	1980	16390	16390	4313	4296*
Double-Precision Implementation						
p	m	n	$\gamma_1 \times \gamma_2 = p \times 1$		$\gamma_1 \times \gamma_2 =$ optimal choice	
			Algorithm I	Hybrid	Algorithm II	Enhanced II
16	500	500	38508	38508	34784	34260*
64	1000	1000	45212	45212	35616	35092*
64	1200	1200	61236	61236	50016	49392*
16	600	400	35756	35756	33660	33060*
64	1200	800	41236	41236	34492	33892*
64	1980	100	8732	8732	8732	8424*
16	400	600	38860	38860	33384*	33410
64	800	1200	46788	46788	34216*	34242
64	100	1980	32780	32780	8626	8592*

Table 3.20: Comparing the enhanced Algorithm II with other schemes.

Chapter 4

Sparse Orthogonal Decomposition on a Hypercube Multiprocessor

4.1 Introduction

Let A be a large sparse $m \times n$ $(m > n)$ matrix with full column rank. We consider the problem of reducing A to upper triangular form using orthogonal transformations on a hypercube multiprocessor. Givens rotations are widely used in the orthogonal decomposition of large sparse matrices on sequential machines [17,25,28,43,50,68]. For parallel computers, Heath and Sorensen [53] have adapted the pipelined Givens algorithm proposed in [13] for implementation on a shared-memory multiprocessor, and they report computational results for the Denelcor HEP computer.

When A is sparse, some zero entries in A may become nonzero during the computing process and therefore appear in the final structure of R. Because these nonzero elements do not exist in A, they are commonly referred to as *fill* in sparse matrix literature. Suppose we apply a Givens rotation to two sparse rows with leading nonzeros in the same column. The first nonzero of one of them will become zero and except for that element, the structure of the two transformed rows becomes the union of their original structures [16,25,43,50,68]. An example is given in Figure 4.1, where the leading nonzero element of the second row is annihilated by the transformation. The first row in Figure 4.1 is said to be the "pivot row" of the applied rotation.

$$\begin{pmatrix} \times & \times & 0 & 0 & \times \\ \times & 0 & \times & 0 & 0 \end{pmatrix} \longrightarrow \begin{pmatrix} \times & \times & \times & 0 & \times \\ 0 & \times & \times & 0 & \times \end{pmatrix}$$

Figure 4.1: Applying a Givens rotation to two sparse rows.

When Givens rotations are applied to a sparse matrix A to reduce it to upper triangular form, we can distinguish two kinds of fill. We explain this using an example. In Figure 4.2 we display the entire process of reducing a 3×3 sparse matrix to its upper triangular form. The zeros in positions (2,1), (3,1) and (3,2) are introduced by applying Givens rotations to the pair {row 1, row 2}, {row 1, row 3} and {row 2, row 3} in order. Referring to

$$\begin{pmatrix} \times & 0 & \times \\ \times & \times & 0 \\ \times & 0 & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & f & \times \\ 0 & \times & f \\ \times & 0 & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & f & \times \\ 0 & \times & f \\ 0 & i & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & f & \times \\ 0 & \times & f \\ 0 & 0 & \times \end{pmatrix}$$

Figure 4.2: Sparse orthogonal reduction by Givens rotations.

Figure 4.2, the (i,j) positions marked "f"are fill in the final structure of R, and the (i,j) position marked "i" denotes an *intermediate* fill which is subsequently annihilated at a later step. The decomposition process above can be viewed as *rotating* or *merging* each row into a gradually computed upper triangular matrix as illustrated in Figure 4.3. Each "row merging" operation annihilates as many nonzero elements as possible in the incoming row. For example, in Figure 4.3 when row 3 is merged into the partially computed upper triangular matrix, two Givens rotations must be applied to annihilate the nonzero in the $(3,1)$ position and the intermediate fill-in thus incurred in the $(3,2)$ position. Since each single row can be viewed as a partially computed upper triangular matrix, in some contexts we refer to the row merging operation as applying a Givens rotation to a pair of rows. Therefore, in general a "row merging" operation may annihilate none, one or more than one nonzero elements depending on the structure of the partially computed R and the structure of the incoming row. The collection of rotations which merge more than one row into a non-null submatrix is termed "submatrix merging" in [61]. The rotation sequence

$$\begin{pmatrix} 0 & 0 & 0 \\ & 0 & 0 \\ & & 0 \\ - & - & - \\ \times & 0 & \times \\ \times & \times & 0 \\ \times & 0 & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & 0 & \times \\ & 0 & 0 \\ & & 0 \\ - & - & - \\ & & \\ \times & \times & 0 \\ \times & 0 & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & f & \times \\ & \times & f \\ & & 0 \\ - & - & - \\ & & \\ & & \\ \times & 0 & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & f & \times \\ & \times & f \\ & & \times \\ & & \\ & & \\ & & \\ & & \end{pmatrix}$$

Figure 4.3: A sequence of row merging operations.

corresponding to the row by row merging process in Figures 4.2 and 4.3 is given in Figure 4.4. Note that if we view row 1 as a submatrix, and view the collection of row 2 and row 3 as

$$\begin{pmatrix} \times & 0 & \times \\ \times & \times & 0 \\ \times & 0 & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & f & \times \\ 1 & \times & f \\ 2 & 3 & \times \end{pmatrix}$$

Figure 4.4: A sparse Givens sequence.

another submatrix, then the rotations 1, 2 and 3 comprise a submatrix merging operation. Rotation 3 is necessary because of the occurrence of the intermediate fill at the (3,2) position when the nonzero element at the (3,1) position was annihilated by rotation 2. Although intermediate fill does not occupy storage in R, it causes higher arithmetic cost. Therefore, the serial sparse Givens algorithms usually aim at reducing both kinds of fill.

In order to obtain a sparse R, George and Heath [28] make use of the following connection between the factor R and the Cholesky factor of A^TA. First, they note that the factor R is mathematically equal to the Cholesky factor of the symmetric positive definite matrix A^TA. Second, they observe that if P_r and P_c are permutation matrices, then

$$(P_rAP_c)^T(P_rAP_c) = P_c^TA^T(P_r^TP_r)(AP_c) = (AP_c)^T(AP_c) = P_c^T(A^TA)P_c\,. \tag{4.1}$$

With these observations, they suggest that a symmetric ordering which produces a sparse Cholesky factor for A^TA also yields an equally sparse R if the permuted matrix AP_c is reduced by orthogonal transformations. Although finding an optimal ordering for a symmetric and positive definite matrix is NP-complete [85], there exist a number of good heuristic orderings which perform well in practice and have efficient implementations [33]. Therefore, it is common practice in sparse matrix computation to subject a given matrix to such an ordering algorithm before determining the structure of R. From now on we shall assume that the columns of the matrix A have been appropriately ordered.

Although it is apparent from (4.1) that the row ordering of A does not have any effect on the sparsity structure of R, it is important in reducing the amount of intermediate fill. The role of row ordering in solving sparse least squares problems has been investigated by George and Ng in [39], and the row-ordering schemes for sparse Givens transformations were studied by George, Liu and Ng in [36,37,38].

The scheme we propose is designed for the parallel implementation of the general row merging scheme developed by Liu [61] for sparse Givens transformations. The row ordering implicitly imposed by the rotation sequence of this scheme appears to introduce significantly less intermediate fill in the process of computing R compared to other known schemes. In Section 2, we review the George-Heath scheme and Liu's general row merging scheme. The parallel version of the latter and its complexity analysis on a model problem are presented in Section 3.

4.2 Serial Row Merging Scheme

One of the main objectives of the serial row merging scheme is to reduce intermediate fill during the process of orthogonal decomposition either via Givens rotations or Householder transformations [34,61]. Since the arithmetic cost of the algorithm is closely related to the intermediate fill during the computation, the reduction in factorization time can be very significant as demonstrated by numerical experiments in [34,61]. It was also shown in [34,61] that the trade-off for the lower computational cost is only a very modest increase in working storage. Working storage is an important consideration for parallel implementation on local-memory machines, because there is relatively much less memory available on each node processor compared to a sequential machine. Since the parallel implementation performs essentially the same computation in a distributed manner, the strategy leading to a reduced amount of arithmetic in the serial algorithm is likely to benefit its parallel implementation as well. A possible reservation is that for some very efficient serial algorithm, it may be difficult to find an efficient parallel implementation due to its inherent serial bottlenecks. Fortunately, as we shall see later, the row merging scheme does exhibit rich parallelism suitable for implementation on a local-memory multiprocessor.

4.2.1 An Example

Following Liu [61], we use the example in Figure 4.5 to demonstrate how the row merging schemes can be understood as generalizing the *row rotations* in the George-Heath scheme [28] to *submatrix rotations*. In what follows, each (i,j) entry corresponding to a nonzero element in the given coefficient matrix is marked "$\times$" and each fill in the triangular factor R is marked "f".

$$\begin{pmatrix}
a_{1,1} & & & & a_{1,5} & & a_{1,7} & a_{1,8} & \\
a_{2,1} & & & & a_{2,5} & & a_{2,7} & a_{2,8} & \\
a_{3,1} & & & & a_{3,5} & & a_{3,7} & a_{3,8} & \\
a_{4,1} & & & & a_{4,5} & & a_{4,7} & a_{4,8} & \\
 & a_{5,2} & & & a_{5,5} & & & a_{5,8} & a_{5,9} \\
 & a_{6,2} & & & a_{6,5} & & & a_{6,8} & a_{6,9} \\
 & a_{7,2} & & & a_{7,5} & & & a_{7,8} & a_{7,9} \\
 & a_{8,2} & & & a_{8,5} & & & a_{8,8} & a_{8,9} \\
 & & a_{9,3} & & & a_{9,6} & a_{9,7} & a_{9,8} & \\
 & & a_{10,3} & & & a_{10,6} & a_{10,7} & a_{10,8} & \\
 & & a_{11,3} & & & a_{11,6} & a_{11,7} & a_{11,8} & \\
 & & a_{12,3} & & & a_{12,6} & a_{12,7} & a_{12,8} & \\
 & & & a_{13,4} & & a_{13,6} & & a_{13,8} & a_{13,9} \\
 & & & a_{14,4} & & a_{14,6} & & a_{14,8} & a_{14,9} \\
 & & & a_{15,4} & & a_{15,6} & & a_{15,8} & a_{15,9} \\
 & & & a_{16,4} & & a_{16,6} & & a_{16,8} & a_{16,9}
\end{pmatrix}$$

Figure 4.5: A 16×9 example.

I. George-Heath Scheme [28]

Step 1. Determine the structure (not the numerical values) of $B = A^T A$.

Step 2. Find an ordering for B (column ordering for A) which has a sparse Cholesky factor R.

Step 3. Symbolically factor the reordered B, generating a row-oriented data structure for R.

Step 4. Compute R by processing the rows of A one by one using Givens rotations.

We assume A has been properly ordered and depict in Figure 4.6 the rotation sequence of Step 4 by numbering the *zeroed* locations in the order they are created. By

$$\begin{pmatrix}
\times & & & & \times & & \times & \times & \\
1 & & & & \times & & \times & \times & f \\
2 & & & & 3 & & \times & \times & f \\
4 & & & & 5 & & 6 & \times & f \\
 & \times & & & \times & & & \times & \times \\
 & 7 & & & 8 & & 9 & 10 & \times \\
 & 11 & & & 12 & & 13 & 14 & 15 \\
 & 16 & & & 17 & & 18 & 19 & 20 \\
 & & \times & & & \times & \times & \times & \\
 & & 21 & & & \times & \times & \times & f \\
 & & 22 & & & 23 & 24 & 25 & 26 \\
 & & 27 & & & 28 & 29 & 30 & 31 \\
 & & & \times & & \times & & \times & \times \\
 & & & 32 & & 33 & 34 & 35 & 36 \\
 & & & 37 & & 38 & 39 & 40 & 41 \\
 & & & 42 & & 43 & 44 & 45 & 46
\end{pmatrix}$$

Figure 4.6: The George-Heath scheme.

comparing the zeroed pattern with the original coefficient matrix, we immediately see that the intermediate fills are in locations {9, 13, 18, 26, 31, 34, 39, 44}. The total

$$\begin{pmatrix}
\times & & & & \times & & \times & \times & \\
 & \times & & & \times & & & \times & \times \\
 & & \times & & & \times & \times & \times & \\
 & & & \times & & \times & & \times & \times \\
 & & & & \times & & \times & \times & f \\
 & & & & & \times & \times & \times & f \\
 & & & & & & \times & \times & f \\
 & & & & & & & \times & f \\
 & & & & & & & & \times
\end{pmatrix}$$

Figure 4.7: The resulting upper triangular factor.

number of rotations required by this scheme is 46. The resulting upper triangular factor is shown in Figure 4.7.

The George-Heath scheme implicitly employs a *fixed* pivoting strategy. The chosen "pivot row" of each rotation which annihilates a nonzero in a particular column is always the same. For example, row 1 in Figure 4.6 is used as the pivot row to annihilate nonzero element $a_{2,1}$ by rotation 1. When row 3 and row 4 are subsequently processed one after another, row 1 is again used as pivot row to zero out $a_{3,1}$ and $a_{4,1}$ because it is the only row which remains in the partially computed R with leading nonzero in the first column.

II. The General Row Merging Scheme [61]

Step 1. Partition the $m \times n$ coefficient matrix into k submatrices, $k \leq n$. Each submatrix consists of all of the rows having their leading nonzeros in the same position. The four submatrices obtained from the given example are separated by dotted lines in Figure 4.8.

Step 2. Apply Givens rotations to each submatrix and carry out the reduction as far as possible.

$$\begin{pmatrix}
a_{1,1} & & & & a_{1,5} & & a_{1,7} & a_{1,8} & \\
a_{2,1} & & & & a_{2,5} & & a_{2,7} & a_{2,8} & \\
a_{3,1} & & & & a_{3,5} & & a_{3,7} & a_{3,8} & \\
a_{4,1} & & & & a_{4,5} & & a_{4,7} & a_{4,8} & \\
- & - & - & - & - & - & - & - & - \\
 & a_{5,2} & & & a_{5,5} & & & a_{5,8} & a_{5,9} \\
 & a_{6,2} & & & a_{6,5} & & & a_{6,8} & a_{6,9} \\
 & a_{7,2} & & & a_{7,5} & & & a_{7,8} & a_{7,9} \\
 & a_{8,2} & & & a_{8,5} & & & a_{8,8} & a_{8,9} \\
- & - & - & - & - & - & - & - & - \\
 & & a_{9,3} & & & a_{9,6} & a_{9,7} & a_{9,8} & \\
 & & a_{10,3} & & & a_{10,6} & a_{10,7} & a_{10,8} & \\
 & & a_{11,3} & & & a_{11,6} & a_{11,7} & a_{11,8} & \\
 & & a_{12,3} & & & a_{12,6} & a_{12,7} & a_{12,8} & \\
- & - & - & - & - & - & - & - & - \\
 & & & a_{13,4} & & a_{13,6} & & a_{13,8} & a_{13,9} \\
 & & & a_{14,4} & & a_{14,6} & & a_{14,8} & a_{14,9} \\
 & & & a_{15,4} & & a_{15,6} & & a_{15,8} & a_{15,9} \\
 & & & a_{16,4} & & a_{16,6} & & a_{16,8} & a_{16,9}
\end{pmatrix}$$

Figure 4.8: The row merging scheme - step 1.

In Figure 4.9, we again number the zeroed locations to illustrate the rotation sequence assuming the submatrices are processed one by one from top to bottom.

$$\left(\begin{array}{ccccccccc}
\times & & & & \times & & \times & \times & \\
1 & & & & \times & & \times & \times & \\
2 & & & & 3 & & \times & \times & \\
4 & & & & 5 & & 6 & \times & \\
\hline
 & \times & & & \times & & & \times & \times \\
 & 7 & & & \times & & & \times & \times \\
 & 8 & & & 9 & & & \times & \times \\
 & 10 & & & 11 & & & 12 & \times \\
\hline
 & & \times & & & \times & \times & \times & \\
 & & 13 & & & \times & \times & \times & \\
 & & 14 & & & 15 & \times & \times & \\
 & & 16 & & & 17 & 18 & \times & \\
\hline
 & & & \times & & \times & & \times & \times \\
 & & & 19 & & \times & & \times & \times \\
 & & & 20 & & 21 & & \times & \times \\
 & & & 22 & & 23 & & 24 & \times
\end{array}\right)$$

Figure 4.9: The row merging scheme - step 2.

Step 3. Obtain a new set of submatrices by ignoring the first row of each reduced submatrix. Merge the submatrices which have the same column subscript for the first nonzero element in their (current) first row.

For example, the two submatrices defined by rows 2, 3, 4 and 6, 7, 8 in Figure 4.9 satisfy this condition. So are the two submatrices defined by rows 10, 11, 12 and rows 14, 15, 16. Therefore each pair of the submatrices will be merged by Givens rotations as depicted in Figure 4.10.

$$\left(\begin{array}{ccccccccc}
x & & & & x & & x & x & \\
 & & & & x & & x & x & f \\
 & & & & & & x & x & f \\
 & & & & & & & x & f \\
 & x & & & x & & & x & x \\
 & & & & 25 & & 26 & 27 & x \\
 & & & & & & & 28 & 29 \\
 & & & & & & & & 30 \\
- & - & - & - & - & - & - & - & - \\
 & & x & & & x & x & x & \\
 & & & & & x & x & x & f \\
 & & & & & & x & x & f \\
 & & & & & & & x & f \\
 & & & x & & x & & x & x \\
 & & & & & 31 & 32 & 33 & x \\
 & & & & & & & 34 & 35 \\
 & & & & & & & & 36
\end{array}\right)$$

Figure 4.10: The row merging scheme - step 3.

The merging process applied to the two pairs of submatrices annihilates locations 25 to 30 for the first pair and locations 31 to 36 for the second pair. The merging process simply annihilates as many nonzeros as possible in the second matrix by Givens rotations. Alternatively, we can view each submatrix merging operation as applying the George-Heath scheme to one submatrix with another upper trapezoidal submatrix being the only computed part of the triangular factor.

Step 3 is repeated for every newly generated set of submatrices until only one submatrix remains. Then the algorithm terminates and we have generated all the rows of the triangular factor R. For the given example, Figure 4.11 illustrates the final step which involves submatrices defined by rows 3, 4, 6 and rows 11, 12, 14 respectively. The rotations annihilate locations 37 to 42.

$$\begin{pmatrix}
\times & & & & \times & & \times & \times & \\
 & & & & \times & & \times & \times & f \\
 & & & & & & \times & \times & f \\
 & & & & & & & \times & f \\
 & \times & & & \times & & & \times & \times \\
 & & & & & & & & \times \\
 & & & & & & & & \\
 & & & & & & & & \\
 & & \times & & & \times & \times & \times & \\
 & & & & & \times & \times & \times & f \\
 & & & & & & 37 & 38 & 39 \\
 & & & & & & & 40 & 41 \\
 & & & \times & & \times & & \times & \times \\
 & & & & & & & & 42 \\
 & & & & & & & & \\
 & & & & & & & &
\end{pmatrix}$$

Figure 4.11: The row merging scheme - the final step.

For this scheme, the intermediate fill correspond to locations {26, 32, 39, 41}. The total number of rotations is 42. Note that the first row of each submatrix obtained from a merging operation is the s^{th} row of the factor R, where s is the column subscript of that row's first nonzero element. The potential for parallel computation of this scheme can be seen by observing that the merging rotations annihilating locations {25, 26, 27, 28, 29, 30} are independent of the rotations annihilating locations {31, 32, 33, 34, 35, 36}, and these two merging processes can be performed concurrently.

Finally the row merging scheme can be viewed as employing a special *variable* pivoting strategy. Referring to Figures 4.7–4.11, observe that in contrast to the fixed pivoting strategy, the choice of "pivot row" may be *different* for each rotation which annihilates nonzeros in the same column. For example, rotation 26 uses row 3 as the pivot row to eliminate a nonzero at the position (6,7), whereas rotation 32 uses row 11 as the pivot row to eliminate a nonzero at the position (14,7). Although the pivot rows vary, the choices are determined nonetheless by the structure of A and the method. Therefore, the pivoting strategy of the row merging scheme can be viewed as a special form of variable pivoting [25,61].

4.2.2 Definitions

In the previous section we used a 16×9 matrix to demonstrate how the row merging scheme works. The same example can also be used to motivate the concept of *row merge tree* and *essentially full submatrices*. Figure 4.9 depicts the remaining nonzero elements at the end of Step 2 of the general row merging scheme. As mentioned earlier, the top row of each submatrix is the s^{th} row of the factor R, where s is the column subscript of this row's first nonzero element. In Figures 4.12 to 4.15 we display the four submatrices in Figure 4.9 separately and explicitly label the final (i,j) entry in R by $r_{i,j}$.

$$\begin{pmatrix} r_{1,1} & 0 & 0 & 0 & r_{1,5} & 0 & r_{1,7} & r_{1,8} & 0 \\ 0 & 0 & 0 & 0 & \times & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & 0 \end{pmatrix}$$

Figure 4.12: The first submatrix.

These four submatrices are each an essentially full upper triangular matrix as defined

$$\begin{pmatrix} 0 & r_{2,2} & 0 & 0 & r_{2,5} & 0 & 0 & r_{2,8} & r_{2,9} \\ 0 & 0 & 0 & 0 & \times & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times \end{pmatrix}$$

Figure 4.13: The second submatrix.

$$\begin{pmatrix} 0 & 0 & r_{3,3} & 0 & 0 & r_{3,6} & r_{3,7} & r_{3,8} & 0 \\ 0 & 0 & 0 & 0 & 0 & \times & \times & \times & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \times & \times & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & 0 \end{pmatrix}$$

Figure 4.14: The third submatrix.

$$\begin{pmatrix} 0 & 0 & 0 & r_{4,4} & 0 & r_{4,6} & 0 & r_{4,8} & r_{4,9} \\ 0 & 0 & 0 & 0 & 0 & \times & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times \end{pmatrix}$$

Figure 4.15: The fourth submatrix.

in [61]. We now state Liu's definition for essentially full upper triangular and trapezoidal matrices.

Definition 1 *Let T be a sparse upper triangular matrix. Suppose the i^{th} row and the i^{th} column of T are removed if they contain only zeros. After removing all null row/column pairs from T, if the remaining matrix is full upper triangular, then T is said to be essentially full upper triangular. If the resulting matrix is full upper trapezoidal, then T is said to be essentially full upper trapezoidal. In either case, the structure of T is fully characterized by the number of remaining rows together with the column subscript set. An essentially full upper triangular or trapezoidal matrix is denoted by $TZ[k, \{j_1, \cdots, j_{\ell-1}, j_\ell\}]$, where k is the number of non-null rows in T, and $\{j_1, \cdots, j_{\ell-1}, j_\ell\}$ are the column subscripts corresponding to the non-null columns in T.*

By Definition 1, the four submatrices in Figures 4.12–4.15 are essentially full and they can be referred to as $TZ[4, \{1, 5, 7, 8\}]$, $TZ[4, \{2, 5, 8, 9\}]$, $TZ[4, \{3, 6, 7, 8\}]$ and $TZ[4, \{4, 6, 8, 9\}]$. As an example we cast the matrix in Figure 4.14 into a 6-by-6 sparse upper triangular matrix in Figure 4.16 to show that it does indeed satisfy Definition 1.

$$\begin{pmatrix} r_{3,3} & 0 & 0 & r_{3,6} & r_{3,7} & r_{3,8} \\ & 0 & 0 & 0 & 0 & 0 \\ & & 0 & 0 & 0 & 0 \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{pmatrix}$$

Figure 4.16: $TZ[4,\{3,6,7,8\}]$.

Since the rows containing $r_{i,j}$ entries do not participate in the subsequent merging process, the submatrices participating in Step 3 are obtained by deleting the top row from each submatrix. The four submatrices so obtained are each still an essentially full upper triangular matrix as displayed in Figures 4.17.

$$\begin{pmatrix} a_{2,5} & a_{2,7} & a_{2,8} \\ & a_{3,7} & a_{3,8} \\ & & a_{4,8} \end{pmatrix} \begin{pmatrix} a_{6,5} & a_{6,8} & a_{6,9} \\ & a_{7,8} & a_{7,9} \\ & & a_{8,9} \end{pmatrix} \begin{pmatrix} a_{10,6} & a_{10,7} & a_{10,8} \\ & a_{11,7} & a_{10,8} \\ & & a_{12,8} \end{pmatrix} \begin{pmatrix} a_{14,6} & a_{14,8} & a_{14,9} \\ & a_{15,8} & a_{14,9} \\ & & a_{16,9} \end{pmatrix}$$

Figure 4.17: TZ[3,{5,7,8}], TZ[3,{5,8,9}], TZ[3,{6,7,8}] and TZ[3,{6,8,9}].

For easy explanation of the concept of row merge tree and its role in guiding the computation in the general row merging scheme, we review the definitions of four trees which are closely related to the row merge tree defined by Liu [61]. They are referred to as elimination tree, row merge tree, binary row merge tree and reduced row merge tree. The definition of row merge tree given in Liu [61] is for our binary row merge tree.

Definition 2 *(Elimination tree of A^TA) Given an $m \times n$ matrix A. Assuming that A^TA is irreducible, the elimination tree of A^TA is a tree consisting of n vertices each being uniquely labelled by an integer in $\{1, \cdots, n-1, n\}$. Let R denote the upper triangular factor from the orthogonal decomposition of A or the Cholesky decomposition of A^TA. If $r_{i,j}$ $(i < j)$ is the leading off-diagonal nonzero in the i^{th} row of R, then vertex j is the parent of vertex i in the elimination tree.*

By Definition 2, the elimination tree corresponding to the 16×9 matrix A given in Figure 4.5 can be immediately obtained from the structure of R given in Figure 4.7. For easy reference, we re-display the structure of R in Figure 4.18 and display the elimination tree associated with A in Figure 4.19.

The elimination tree of a square sparse matrix M has been used to set up efficient data structures and to guide serial and parallel computation in factoring M via Gaussian elimination [14,15,29,31,33,57,59,60,70,80].

$$\begin{pmatrix}
r_{1,1} & & & & r_{1,5} & & r_{1,7} & r_{1,8} & \\
& r_{2,2} & & & r_{2,5} & & & r_{2,8} & r_{2,9} \\
& & r_{3,3} & & & r_{3,6} & r_{3,7} & r_{3,8} & \\
& & & r_{4,4} & & r_{4,6} & & r_{4,8} & r_{4,9} \\
& & & & r_{5,5} & & r_{5,7} & r_{5,8} & r_{5,9} \\
& & & & & r_{6,6} & r_{6,7} & r_{6,8} & r_{6,9} \\
& & & & & & r_{7,7} & r_{7,8} & r_{7,9} \\
& & & & & & & r_{8,8} & r_{8,9} \\
& & & & & & & & r_{9,9}
\end{pmatrix}$$

Figure 4.18: The factor R from Figure 4.7.

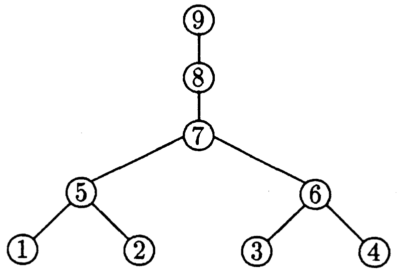


Figure 4.19: The elimination tree associated with the matrix A in Figure 4.5.

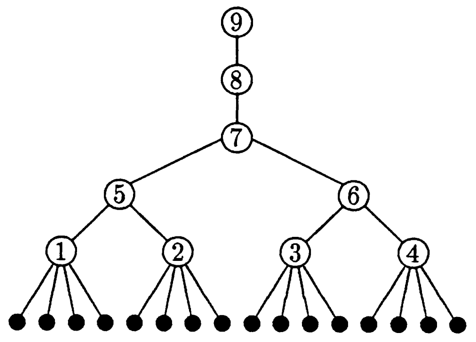


Figure 4.20: The row merge tree associated with the matrix A in Figure 4.5.

To obtain the row merge tree of an $m \times n$ matrix A, we add m leaves to the associated elimination tree in the following manner. If A has m_i rows with leading nonzeros in the i^{th} column, m_i leaves are attached to vertex i in the corresponding elimination tree. For the matrix A in Figure 4.5, we obtain the row merge tree as given in Figure 4.20.

Now, if the resulting row merge tree is not binary, it can be transformed into a binary tree by introducing additional interior nodes (binary splitting) if a node has more than two children, and by removing those parent nodes that have only one child [61]. We define the transformed binary tree to be A's binary row merge tree. We demonstrate this process by transforming the row merge tree in Figure 4.20 to the binary row merge tree in Figure 4.21. Therefore, for a given $m \times n$ matrix, its binary row merge tree is a strictly binary tree with m leaves, each corresponding to a row in the matrix, which is the definition given in [61] for row merge tree. As pointed out by Liu in [61], there are different ways to perform binary-splitting of the row merge tree. To find the best possible splitting in the context of sparse QR is a research problem in its own right. More on the splitting criterion and strategies can be found in [16,61,87].

The name "row merge tree" is coined in [61] based on the following observation. Refer-

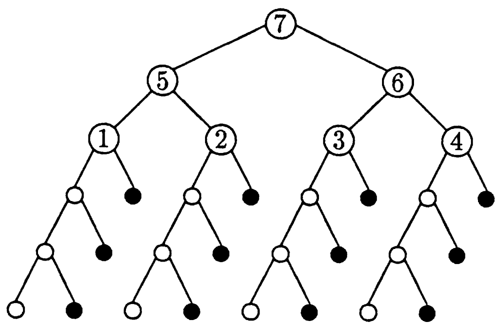


Figure 4.21: The binary row merge tree associated with the matrix A in Figure 4.5.

ring to the binary row merge tree in Figure 4.21, every *interior* vertex v_i defines a subtree rooted at itself. From the definition of the binary row merge tree, the leaves of each subtree rooted at an interior vertex represent a subset of rows from the coefficient matrix A. Therefore, one can associate with each interior vertex an upper triangular matrix obtained by the orthogonal reduction of the corresponding rows in its subtree. Using the definition for essentially full upper triangular or trapezoidal matrices, for the given example the submatrix associated with vertex $v_i = 1$ is $TZ[4, \{1, 5, 7, 8\}]$, and the submatrix associated with vertex $v_i = 2$ is $TZ[4, \{2, 5, 8, 9\}]$. The submatrix associated with vertex $v_i = 5$ is then obtained by merging the two submatrices associated with vertices $v_i = 1$ and $v_i = 2$. Clearly the matrix associated with the root of the binary row merge tree is the triangular factor R.

From an implementation point of view, the following further observations are the keys to the successful use of the full matrix technique which is to be explained in our discussion below. They are also crucial in understanding the algorithmic description of the row merging scheme presented in the next section. First, note that there is a one-to-one mapping between each row in the factor R and each vertex in the elimination tree. Recall that after each submatrix merging operation, the top row of the resulting submatrix will not participate

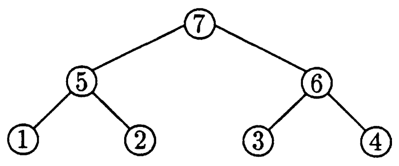


Figure 4.22: The reduced row merge tree of A in Figure 4.5.

in any one of the future reduction steps. Using the row merge tree terminology, the s^{th} row of R is computed by the submatrix merging operation associated with vertex $v_i = s$ in the elimination tree. Observe that there is a chain of vertices $v_i = 7$, 8 and 9 in the elimination tree. This indicates that row 7, row 8 and row 9 of R are all computed by the single submatrix merging operation associated with the vertex at which the chain begins, i.e., $v_i = 7$ in this case. Therefore, in terms of defining "submatrix merging operations", we can simply look at a "reduced row merge tree" which is transformed from the elimination tree by removing those parent nodes which have only one child. See Figure 4.22 for the reduced row merge tree of the same example.

Second, if each completely computed row of R is immediately stored into the static data structure set up separately for the factor R, then such a row can be *deleted* from the submatrix containing it. Using our example, for $v_i = 1$, row 1 of R is computed and deleted from $TZ[4, \{1,5,7,8\}]$; for $v_i = 2$, row 2 of R is computed and deleted from $TZ[4, \{2,5,8,9\}]$. The remaining rows form submatrices $TZ[3, \{5,7,8\}]$ and $TZ[3, \{5,8,9\}]$. The merging of the two can be done by allocating storage (dynamically from a stack) for one $k \times k$ *full* upper triangular matrix, where $k = 4$ is the size of the union of the two column subscript sets, i.e., $\{5,7,8,9\}$. We demonstrate this process in Figure 4.23. Since the merging operation computes row 5 of R, the subscript set $\{5,7,8,9\}$ is also known from the nonzero structure of R, which is available after $A^T A$ is symbolically factored.

$$\begin{pmatrix} \times & \times & \times & \\ & \times & \times & \\ & & \times & \\ - & - & - & - \\ \times & & \times & \times \\ & & \times & \times \\ & & & \times \end{pmatrix} \longrightarrow \begin{pmatrix} r_{5,5} & r_{5,7} & r_{5,8} & r_{5,9} \\ & \times & \times & \times \\ & & \times & \times \\ & & & \times \\ & & & \\ & & & \\ & & & \end{pmatrix}$$

Figure 4.23: Applying full matrix technique to submatrix merging.

If we preprocess the rows of A as described by Step 1 and Step 2 of the general row merging scheme, initially the submatrices are available at each *leaf* node of the reduced row merge tree. Since the task associated with each interior node of the reduced row merge tree is to merge the two submatrices associated with its two children *after* they are formed, it is desirable to find an ordering to perform these tasks so that the submatrices are always formed before they are needed and they are conveniently accessible whenever they are needed. The post-ordering traversal of the reduced row merge tree generates such a sequence, because it ensures that the children are always visited before their parent is visited in the traversal. For example, a post-ordering traversal of the tree in Figure 4.22 consists of visiting vertices 1, 2, 5, 3, 4, 6 and 7 in order. The overall organization of the computation makes use of a stack and is presented in the next section.

As far as generating the row merge tree is concerned, we simply recall that the nonzero structure of the factor R is available after symbolic factorization of $A^T A$. Alternatively, the structure of R can be generated directly from A using a symbolic submatrix merging algorithm described in [61]. More details about the properties of the row merge tree and other related work is provided in [61]. For our purpose, it is essential to understand how the row merge tree induces a computational sequence for performing Givens rotations in the serial row merging scheme, and what role the row merge tree can play to identify and exploit parallelism in the parallel row merging scheme we propose in this chapter.

4.2.3 The Algorithm

For completeness we restate the sequential row merging algorithm, which was described by Liu in [61].

Step 1. (Column Ordering) Find a fill-reducing ordering P_c (e.g., a minimum degree ordering) for the structure of $A^T A$. Obtain the structure of AP_c using P_c as the column ordering.

Step 2. (Symbolic Factorization) Perform the symbolic factorization and generate the (not necessarily binary) row merge tree.

Step 3. (Row Ordering) Traverse the row merge tree by a depth-first search and record the post-ordering in a linear array $LABEL$. Sort the rows of AP_c according to their first column subscripts, following the order given by $LABEL$. That is, the rows with their first column subscript equal to $LABEL[i]$ should come before the rows with their first column subscript equal to $LABEL[j]$ if $i < j$. Let the corresponding row ordering be P_r.

Step 4. (Numeric Factorization) Initialize a stack of essentially full upper trapezoidal matrices.

```
for $i := 1$ to $n$
begin
4.1 From the structure of $R$, let $LABEL[i]$, $i_1$, $i_2$, ..., $i_t$, be the locations of the
    nonzeros in row $R_{LABEL[i]*}$.
4.2 Obtain working space for a full $(t+1)$ by $(t+1)$ upper triangular matrix.
```

```
4.3 If the top upper trapezoidal matrix on the stack has $LABEL[i]$ as its first sub-
    script then Pop it from the stack and merge it into the working triangular matrix.
4.4 While the next row from $P_rAP_c$ has $LABEL[i]$ as its first subscript, merge the
    row into the working triangular matrix.
4.5 Save the first row of working triangular matrix as the row $R_{LABEL[i]*}$ into the
    static storage set up for $R$.
4.6 Consider the remaining non-null rows of working matrix as one upper trapezoidal
    matrix with the subscript set: $i_1$, ..., $i_t$.
4.7 If the top upper triangular matrix on the stack has $i_1$ as its first subscript then
    Merge the remaining working matrix into this top matrix else Push the remaining
    working matrix onto the stack.
end
```

In this description of the serial row merging scheme, the array $LABEL$ records the post-ordering of the nodes of the row merge tree. Associated with each node $s = LABEL[i]$ is an essentially full trapezoidal submatrix denoted by $TZ[u_s,U_s]$. (Although it is possible that some upper trapezoidal submatrices may not be essentially full, the bookkeeping overhead involved in further exploiting this level of sparsity is not considered justified [61].) Since the column subscript set U_s can be identified as the row structure of R_{s*}, it is not necessary to store it explicitly.

The working storage, the stack size, and the amount of computation involved in each merging process associated with an interior node $LABEL[i]$ can also be determined based on the row merge tree, the structure of the matrix P_rAP_c, and the structure of the factor R. The details of these aspects were examined in [61].

4.3 Parallel Row Merging Scheme

We now consider implementing the row merging scheme on a hypercube multiprocessor. Since the column ordering step, symbolic factorization step and the row ordering step are each a research topic worthy of separate investigation, we shall consider only the problem of parallelizing the numeric factorization phase described by Step 4. The proposed parallel algorithm is designed to perform the numeric factorization on a hypercube machine with $p = 2^d$ processors, where d is the dimension of the hypercube network. We assume that at the end of step 3, the integer array $LABEL$, the permuted matrix $P_r A P_c$, and the structure of the triangular factor R are all available in the host. Since there is no globally shared memory among the p processing nodes, or between the host and a node processor, the data must be distributed among the processors in some way, and the mapping strategy should be devised to maintain high parallelism throughout the computation. Since the entire computing process is a sequence of submatrix merging operations, the reduced row merge tree is the most convenient form to use in our following discussions. Therefore, we shall adopt the convention of using "row merge tree" to imply the reduced row merge tree unless we explicitly state otherwise.

4.3.1 Basic Mapping Considerations

The mapping of data and computing tasks to processors must be done in a manner consistent with the precedence relationship inherent in the algorithm. First, the precedence relationship defined by the row merge tree requires that the computation associated with the children nodes be completed before the work associated with the parent node is completed, and the task associated with the parent node is started after the tasks associated with both of the children nodes are started. There are a number of ways to traverse the tree so that the required precedence relationship is observed, while maintaining the same amount of intermediate fill during the factorization process. The depth first traversal used in Liu's serial row merging scheme is one alternative. Whether the row ordering induced by the depth first traversal is suitable for parallel implementation depends on several issues discussed below.

If we consider the submatrix merging operation associated with each node of the row merge tree as a *computational task*, the *task granularity* is one major consideration in deciding whether each task should be allocated to more than one processor. If the amount of computation demanded by each task is more or less the same, it is possible to obtain *speed-up* by allocating each task to one processor and exploiting the parallelism permitted by the precedence relationship only. This approach has been adopted in [31] to perform sparse Cholesky decomposition on a local-memory multiprocessor, and in [15] to parallelize the multifrontal schemes for shared-memory multiprocessors. However, if the amount of computation demanded by any one task is of the same order of magnitude as the serial algorithm, then regardless of how many processors are available the parallel algorithm using this "one task to one processor" approach can at best be as fast as its serial counterpart (in the order of magnitude sense). In addition, it is not uncommon that each node processor of a local-memory machine has relatively limited memory compared to a sequential machine. Therefore, the above "one task to one processor" approach could also impose a severe limitation on the size of the problem the parallel algorithm can handle on a local-memory multiprocessor.

In [61], the serial row merging algorithm is analyzed for a k-by-k grid model problem. The complexity results given there indicate that the serial time (in terms of multiplicative floating-point operations) is $O(k^3)$. It is well known that when the vertices of a k-by-k regular grid are ordered by the nested dissection method [27], the computation associated with the root of the row merge tree involves the merging of two *full* $k \times k$ upper triangular matrices. Therefore, the serial time for this task alone will also be $O(k^3)$. This is exactly the scenario we are concerned with in the last paragraph. Since there are usually non-trivial communication cost involved when multiple processors on a local-memory machine cooperate to perform a task, the strategy of dividing one task among multiple processors should be used judiciously, probably to a subset of large tasks only. We shall therefore adopt a mixed strategy, namely that we may allocate one processor to handle multiple small tasks, but allocate multiple processors to handle each large-enough task. More will be said later about how such a mixed strategy can be used beneficially in the proposed "parallel row merging scheme".

Another important issue is how to exploit the parallelism permitted by the precedence relationship, which is in turn induced by the row merge tree. For the parallel algorithm we shall propose, a stronger precedence relationship is actually imposed, namely that the task associated with a parent node will start only after the tasks associated with all of the children nodes are finished. It should be pointed out that with the decision made on dividing each task among multiple processors, the stronger precedence relationship does not necessarily imply reduced parallelism. On the contrary, this assumption will simplify the other aspects of the parallel algorithm and could enhance its performance. This can be easily understood from an example. Let us consider the precedence relationship induced by a degenerate tree, say a chain of n nodes. Using the "one task to one processor" approach together with the stronger precedence relationship, the m tasks must be processed sequentially. The parallel algorithm becomes a distributed serial algorithm. However, using the "one task divided among p processors" approach, the parallel algorithm still has the potential to be "p" times faster than the serial algorithm, regardless of how we interpret the precedence relationship induced by the chain of n nodes. In the situation when multiple tasks associated with a subtree are assigned to one processor, it is obvious that these tasks can only be processed sequentially. That is, the precedence relationship induced by the subtree corresponds to the stronger interpretation.

4.3.2 A Parallel Submatrix Merging Algorithm

We now take a closer look at the mixed mapping strategy we proposed in the last section. In our application of the "multiple tasks to one processor" strategy, each chosen group of tasks is associated with a subtree of the row merge tree. The subtrees will be selected so that as many processors as possible can work in parallel. The processor responsible for the tasks associated with a subtree can be viewed as executing the serial row merging algorithm for a smaller problem defined by the subtree. It is clear that if j subtrees each induces same amount of work can be processed simultaneously by j processors in parallel, then a j fold speed-up will result because no communication is necessary among the cooperating processors. The remaining question is how to identify such subtrees. We address this problem later.

There are two crucial decisions to be made with respect to the implementation of the "one task divided among q processors" strategy, namely, how to map data among the q processors and how to embed an efficient communication topology in the hypercube connection network provided for this subset of q processors. Recall that a computational task is defined to be a submatrix merging operation. Therefore, the two essentially full upper trapezoidal submatrices are the data to be divided among the q processors. We shall first examine two variants of the sequential Givens algorithm and choose one which is more suitable for the parallel implementation.

(1) **Method A - Standard Givens Rotations** Without loss of generality, we shall present the method by applying it to the merging of two full $k \times k$ upper triangular matrices R and $\tilde{R}$. To annihilate row $\tilde{r}_{i*}$ from $\tilde{R}$, the sequential algorithm does the following.

```
for $\ell = i, i+1, \cdots, k$ do
    if $|\tilde{r}_{i,\ell}| \geq |r_{\ell,\ell}|$ then
        $t \leftarrow |r_{\ell,\ell}|/|\tilde{r}_{i,\ell}|$
        $s \leftarrow 1/\sqrt{1+t^2}$
        $c \leftarrow st$
    else
        $t \leftarrow |\tilde{r}_{i,\ell}|/|r_{\ell,\ell}|$
        $c \leftarrow 1/\sqrt{1+t^2}$
        $s \leftarrow ct$
    for $j = \ell, \ell+1, \cdots, k$ do
        $v \leftarrow r_{\ell,j}$
        $w \leftarrow \tilde{r}_{i,j}$
        $r_{\ell,j} \leftarrow cv + sw$
        $\tilde{r}_{i,j} \leftarrow -sv + cw$
```

To annihilate row 1 to row k of the matrix $\tilde{R}$ in series, the arithmetic cost (in terms of multiplicative operations) is $\frac{2}{3}k^3 + 2k^2 + \frac{4}{3}k$.

(2) **Method B - Pairwise Givens Rotations** In Method B, we pair the i^{th} row of R and the i^{th} row of $\tilde{R}$ for $1 \leq i \leq k$. We then apply the above algorithm with $l = i$ to each pair of the i^{th} rows, resulting in annihilating the first nonzero value in row $\tilde{r}_{i,*}$. We now have k updated rows in R and $(k-1)$ updated rows in $\tilde{R}$. (The k^{th} row of $\tilde{R}$ has been eliminated.) The rows with their first nonzero elements in identical positions are again paired together, and we apply the above algorithm with $l = i$

to the i^{th} pair of rows, where $2 \leq i \leq k$. This is repeated for $(k-2)$ more times to eliminate the remaining $(k-2)$ rows from $\tilde{R}$. The arithmetic cost (in terms of multiplicative operations) is the same as Method A, namely $\frac{2}{3}k^3 + 2k^2 + \frac{4}{3}k$.

In Method B, the q row merging operations corresponding to the q pairs of rows can be done simultaneously by q available processors. By distributing the rows of the two submatrices over a loop of q processors in a wrap-around fashion, and requiring each processor to send the reduced row to its right neighbour after each row merging operation, we obtain a parallel algorithm to perform the submatrix merging operation on a loop of q processors. We demonstrate how the proposed algorithm works by applying it to the following example.

We consider merging two essentially full upper triangular matrices X and Z on a loop of four processors P_1, P_2, P_3 and P_4, where $X = TZ[6,\{1,2,3,4,5,6\}]$ and $Z = TZ[6,\{1,2,3,4,7,8\}]$. The 4-processor loop and the matrices X and Z are displayed in Figures 4.24 to 4.26.

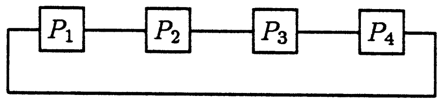


Figure 4.24: A loop of four processors.

The rows of each submatrix are distributed among the processors as shown in Figure 4.27. We note that the wrap mapping is applied to the union of the two column subscript sets, i.e. $\{1,2,3,4,5,6,7,8\}$. If row i is missing from a submatrix, we simply skip the assignment of row i for that submatrix.

As mentioned above, we can view the submatrix merging operation as annihilating as many nonzero elements in the matrix Z as possible via a sequence of Givens rotations. After each processor applies a Givens rotation to the first pair of rows and sends the reduced row to its right neighbour, the updated rows are now distributed among the processors as shown in Figure 4.28. Note that P_4 's right neighbour is P_1, and P_i 's right neighbour is P_{i+1} for

$$\begin{pmatrix}
x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} & x_{1,6} & 0 & 0 \\
 & x_{2,2} & x_{2,3} & x_{2,4} & x_{2,5} & x_{2,6} & 0 & 0 \\
 & & x_{3,3} & x_{3,4} & x_{3,5} & x_{3,6} & 0 & 0 \\
 & & & x_{4,4} & x_{4,5} & x_{4,6} & 0 & 0 \\
 & & & & x_{5,5} & x_{5,6} & 0 & 0 \\
 & & & & & x_{6,6} & 0 & 0 \\
 & & & & & & 0 & 0 \\
 & & & & & & & 0
\end{pmatrix}$$

Figure 4.25: $X = TZ[6,\{1,2,3,4,5,6\}]$.

$$\begin{pmatrix}
z_{1,1} & z_{1,2} & z_{1,3} & z_{1,4} & 0 & 0 & z_{1,7} & z_{1,8} \\
 & z_{2,2} & z_{2,3} & z_{2,4} & 0 & 0 & z_{2,7} & z_{2,8} \\
 & & z_{3,3} & z_{3,4} & 0 & 0 & z_{3,7} & z_{3,8} \\
 & & & z_{4,4} & 0 & 0 & z_{4,7} & z_{4,8} \\
 & & & & 0 & 0 & 0 & 0 \\
 & & & & & 0 & 0 & 0 \\
 & & & & & & z_{7,7} & z_{7,8} \\
 & & & & & & & z_{8,8}
\end{pmatrix}$$

Figure 4.26: $Z = TZ[6,\{1,2,3,4,7,8\}]$.

	P_1	P_2	P_3	P_4
X	*rows* 1, 5	*rows* 2, 6	*row* 3	*row* 4
Z	*row* 1	*row* 2	*rows* 3, 7	*rows* 4, 8

Figure 4.27: Wrap mapping X and Z to a loop of four processors.

$i = 1$ to 3.

	P_1	P_2	P_3	P_4
X	*rows* 1, 5	*rows* 2, 6	*row* 3	*row* 4
Z	*row* 5	*row* 2	*rows* 3, 7	*rows* 4, 8

Figure 4.28: A snap shot.

Our discussion above implies that it is really not necessary to distinguish the origin of each row. We simply pair the rows with identical column subscripts for their leading nonzero elements, and reduce one row using another row via Givens rotations. We therefore distinguish the rows by the column subscripts of their first nonzero elements, regardless of whether they originate from X or Z. We summarize in Figure 4.29 the whole process of merging X and Z into an 8×8 essentially full upper triangular matrix $R = TZ[8, \{1, 2, 3, 4, 5, 6, 7, 8\}]$. We list the rows each processor will have for each step, and it is understood that the action involved in each step is to merge a pair of rows and send the reduced row to the processor's right neighbour in the ring. The "blank" space corresponding to step i and P_j indicates that no row merging operation is performed by processor P_j during step i.

4.3.3 Hypercube Partitioning

Recall that associated with each vertex of the row merge tree is a task involving merging two essentially full upper trapezoidal submatrices. In the previous section we propose a parallel algorithm to divide such a task among a number of processors which form a loop. When the precedence relationship induced by the row merge tree allows us to process several tasks in parallel, it is desirable to partition the available processors into several loops – one for each task. Furthermore, if these tasks have different computational demand, it is also desirable to assign more processors to a bigger task and fewer processors to a smaller task so that the work can be divided evenly among all processors. Therefore, before we can lay out the

	P_1	P_2	P_3	P_4
step 0	1,1,5	2,2,6	3,3,7	4,4,8
step 1	1,5,5	2,2,6	3,3,7	4,4,8
step 2	1,5,5	2,6,6	3,3,7	4,4,8
step 3	1,5,5	2,6,6	3,7,7	4,4,8
step 4	1,5,5	2,6,6	3,7,7	4,8,8
step 5	1,5	2,6,6	3,7,7	4,8,8
step 6		2,6	3,7,7	4,8,8
step 7			3,7	4,8,8
step 8				4,8

Figure 4.29: The entire submatrix merging process.

overall strategy, an important question is "Given an arbitrary number q, does there always exist a subset of q processors in the hypercube machine so that a loop can be embedded?" Although the answer to this question is negative, we have an affirmative answer if q is an even number. With this very mild restriction, we can actually obtain much stronger results which turn out to be very important for the performance of the proposed parallel algorithm. We establish these results in the following theorem.

Theorem 4.1 *Suppose we are given a hypercube connection network with $p = 2^d$ processors. If it is desirable to partition the set of p processors into k disjoint subsets S_1, S_2, ..., and S_k such that*

$$p = \sum_{i=1}^{k} |S_i|$$

and

$$|S_i| = 2\ell_i, \text{ where } \ell_i \text{ is a positive integer},$$

then there exists a (possibly different) partition which maintains the cardinality of each subset, and permits the embedding of k disjoint loops, one for each subset. For each $\ell_i = 1$

we have a degenerate loop consisting of two processors.

Proof: There is a unique mapping from the d-bit *reflected binary Gray code* [74] to the 2^d processor id's of the given hypercube machine, and it is known that the former coding embeds a loop on the hypercube network. Furthermore, any two processors whose id's are different in 1 bit, regardless of the bit position, are connected by a direct link in a hypercube network. By the definition of the reflected binary Gray code, if we represent the 2^{d-1} $(d-1)$-bit Gray code by the array

$$G(d-1) = \{G_0, G_1, G_2, ..., G_{2^{d-1}-1}\} ,$$

then the 2^d d-bit Gray code can be defined recursively by the following equation

$$G(d) = \{0G_0, 0G_1, 0G_2, ..., 0G_{2^{d-1}-1}, 1G_{2^{d-1}-1}, ..., 1G_2, 1G_1, 1G_0\} .$$

Thus any two Gray codes in symmetric positions from the left end and the right end of the array $G(d)$ also differ in one bit only. Therefore, the ℓ_1 processors from the left and of the array and the ℓ_1 processors from the right end of the array form a loop of $2\ell_1$ processors. The $2\ell_2$ processors for S_2 can be chosen from the remaining processors in exactly the same manner, and so on for the $2\ell_i$ processors for the subsets S_i, $3 \leq i \leq k$. This proves the theorem. □

The implication of Theorem 4.1 is in essence that it is not only possible to assign processor loops of different size to handle independent tasks which demand different amounts of computation, but it is also feasible to have all of the processor loops operating simultaneously. Using a hypercube of dimension 4, we illustrate one such partitioning in Figure 4.30.

4.3.4 A Mapping Example

A k-by-k Grid Model Problem

Before we proceed further, let us fix in mind the basic ideas discussed so far by applying them to a k-by-k regular grid problem. This class of problems arises typically in the natural factor

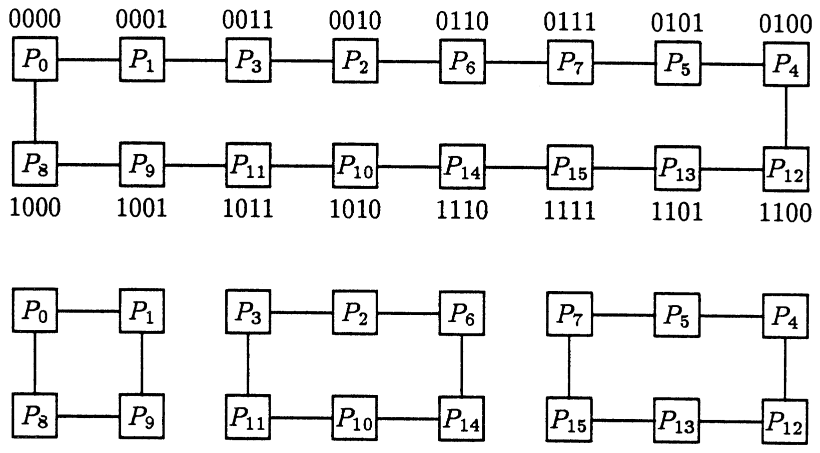


Figure 4.30: Embedding loop(s) in a hypercube of dimension 4.

formulation of the finite element method [1]. Associated with each of the k^2 grid vertices is a variable x_i, where i is the label of the vertex under a chosen ordering scheme. Associated with each of the $(k-1)^2$ small squares are s equations involving the four variables at the corners of the square. Thus, the coefficient matrix of the resulting overdetermined system is $(m = s(k-1)^2)$ by $(n = k^2)$. The example in Figure 4.31 is a 3-by-3 grid with vertices numbered by a nested dissection ordering [27]. When $s = 4$, associated with the grid is the 16×9 matrix given in Figure 4.32.

There is an intimate relationship between the ordering of the vertices and the structure of the elimination tree because the former amounts to permuting the columns of the coefficient matrix and thus determines the sparsity structure of R, which in turn determines the structure of the elimination tree. The ordering schemes which generate an elimination tree with minimum or near-minimum height were examined in [63]. We thus explain first the relationship between the nested dissection ordering and the resulting elimination tree.

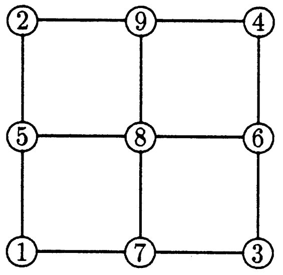


Figure 4.31: Nested dissection ordering of a 3-by-3 grid.

$$
\begin{pmatrix}
a_{1,1} & & & & a_{1,5} & & a_{1,7} & a_{1,8} & \\
a_{2,1} & & & & a_{2,5} & & a_{2,7} & a_{2,8} & \\
a_{3,1} & & & & a_{3,5} & & a_{3,7} & a_{3,8} & \\
a_{4,1} & & & & a_{4,5} & & a_{4,7} & a_{4,8} & \\
 & a_{5,2} & & & a_{5,5} & & & a_{5,8} & a_{5,9} \\
 & a_{6,2} & & & a_{6,5} & & & a_{6,8} & a_{6,9} \\
 & a_{7,2} & & & a_{7,5} & & & a_{7,8} & a_{7,9} \\
 & a_{8,2} & & & a_{8,5} & & & a_{8,8} & a_{8,9} \\
 & & a_{9,3} & & & a_{9,6} & a_{9,7} & a_{9,8} & \\
 & & a_{10,3} & & & a_{10,6} & a_{10,7} & a_{10,8} & \\
 & & a_{11,3} & & & a_{11,6} & a_{11,7} & a_{11,8} & \\
 & & a_{12,3} & & & a_{12,6} & a_{12,7} & a_{12,8} & \\
 & & & a_{13,4} & & a_{13,6} & & a_{13,8} & a_{13,9} \\
 & & & a_{14,4} & & a_{14,6} & & a_{14,8} & a_{14,9} \\
 & & & a_{15,4} & & a_{15,6} & & a_{15,8} & a_{15,9} \\
 & & & a_{16,4} & & a_{16,6} & & a_{16,8} & a_{16,9}
\end{pmatrix}
$$

Figure 4.32: The 16×9 matrix associated with the grid in Figure 4.31.

The Nested Dissection Method and the Elimination Tree

When being applied to a regular grid, the nested dissection method can be understood as choosing a group of vertices as a *separator* to partition the grid recursively. The vertices in the separator are always numbered after the vertices in the two disjoint subgrids are numbered, and the nodes consisting of a separator form a chain in the elimination tree. We show in Figure 4.33 how to generate a nested dissection ordering by recursively defining separators on a 7-by-7 grid. The elimination tree corresponding to the grid in Figure 4.33 is displayed in Figure 4.34. In Figure 4.33, vertices 43 to 49 form the separator S_1^0, which partitions the 7-by-7 grid into two 7-by-3 subgrids. Note that the superscript j in our separator notation S_i^j indicates that there are 2^j separators at this level of recursive partitioning and the subscript i in $\{1, 2, \cdots, 2^j\}$ enumerates them. The vertices 37 to 39 form the separator S_1^1, and the vertices 40 to 42 form the separator S_2^1. S_1^1 and S_2^1 partition the two 7-by-3 subgrids into four 3-by-3 subgrids.

21	25	22	43	30	34	31
23	26	24	44	32	35	33
19	27	20	45	28	36	29
37	38	39	46	42	41	40
3	7	4	47	12	16	13
5	8	6	48	14	17	15
1	9	2	49	10	18	11

21	S_2^2	22	S_1^0	30	S_4^2	31
S_5^3	S_2^2	S_6^3	S_1^0	S_7^3	S_4^2	S_8^3
19	S_2^2	20	S_1^0	28	S_4^2	29
S_1^1	S_1^1	S_1^1	S_1^0	S_2^1	S_2^1	S_2^1
3	S_1^2	4	S_1^0	12	S_3^2	13
S_1^3	S_1^2	S_2^3	S_1^0	S_3^3	S_3^2	S_4^3
1	S_1^2	2	S_1^0	10	S_3^2	11

Figure 4.33: Nested dissection ordering of a 7-by-7 grid and separators.

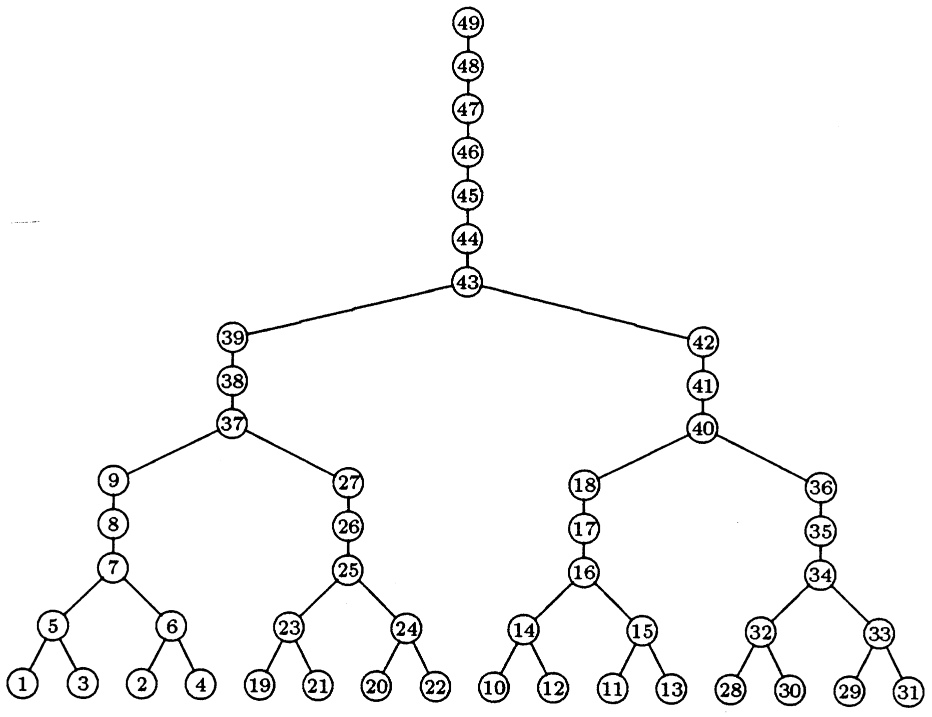


Figure 4.34: An example of an elimination tree.

The Application of a Mixed Mapping Strategy

Suppose we are given a hypercube of dimension 3, i.e., there are eight processors available. Using the elimination tree in Figure 4.34 as an example, our discussion so far suggests the following mapping. For the eight independent subtrees rooted at nodes 5, 6, 23, 24, 14, 15, 32 and 33, each will be assigned to one processor. The four tasks associated with nodes 7, 25, 16 and 34 will each be handled by two processors. Recall that a chain of nodes in the elimination tree simply means no merging is necessary for the nodes other than the first one. In the next level, the two tasks associated with nodes 37 and 40 will each be handled by a loop of 4 processors. Finally the merging of two 7-by-7 full upper triangular matrices, which is the task associated with node 43, will be handled by a loop of 8 processors. For the k-by-k grid problem, all tasks corresponding to the nodes at the same level of the elimination tree are of the same size (strictly speaking, this is true when $k = 2^l - 1$, $l > 0$), and the tasks are getting larger as we move up toward the root of the tree. Therefore, the above mapping does actually achieve the dual goals of assigning multiple small tasks to one processor and dividing each larger task among multiple processors.

In addition, this mixed mapping strategy also leads to the following very favourable situation. For a given machine with p processors, the larger the value of k, the larger the number of nodes (or tasks) in each of the p independent subtrees, and the larger the tasks which are each handled by multiple processors. As noted earlier, the speed-up obtained by processing independent subtrees in parallel is the best one can expect, because there is no communication overhead involved.

4.3.5 Complexity Analysis of a Model Problem

In this section, we shall give a complexity analysis of the parallel row merging scheme applied to a k-by-k grid model problem. This class of problems is well-suited to our approach, and serves to demonstrate the potential speed-up and the effectiveness of the communication scheme supported by the embedded multi-loop topology.

We first note that the nested dissection ordering [27] fits our mapping strategy very well in terms of minimizing communication cost and balancing work load. The match is not accidental, because it hinges on the recursive nature of the nested dissection method coupled with the recursive structure of the hypercube network and the recursive characteristic of the multi-loop embedding scheme we propose in Theorem 4.1. Recall that the elimination tree in Figure 4.34 is induced by the nested dissection numbering of a 7-by-7 grid. Here we shall use the same tree again to demonstrate the intimate relationship among the three. Let us use a hypercube of dimension 3 as an example. Given below is the reflected binary Gray code for the eight 3-bit processor id's.

$$G(3) = \{000, 001, 011, 010, 110, 111, 101, 100\}\,.$$

Let us represent the set of processors assigned to perform the task associated with node v_i by a linear array $\mathrm{L}_{v_i}(N)$, where N is the number of processors. Using natural numbers, we can enumerate the processors in the 8-loop assigned to the task node 43 of the elimination tree as

$$\mathrm{L}_{43}(8) = \{P_0, P_1, P_3, P_2, P_6, P_7, P_5, P_4\}.$$

Following Theorem 4.1, the two 4-loops employed by node 37 and node 40 are

$$\mathrm{L}_{37}(4) = \{P_0, P_1, P_5, P_4\}$$

and

$$\mathrm{L}_{40}(4) = \{P_3, P_2, P_6, P_7\}\,.$$

We observe the following:

1. Since the loops assigned to the task nodes at the same level of the row merge tree form a partition of the hypercube network, they are disjoint sets of processors and there is no competition for communication channels among the loops.

2. Suppose we designate the left-most processor in each linear array $\mathrm{L}_{v_i}(N)$ as the leading processor for the associated task node. For example, the leading processors for the task nodes 43, 37 and 40 are P_0, P_0 and P_3 respectively. Suppose that the leading processor for each task node is responsible for receiving submatrices to be merged and sending the reduced submatrix to the leading processor for the parent task node. Since the row merge tree induced by the nested dissection ordering on a k-by-k grid is binary, completely balanced, and the computational work load is the same for all task nodes at the same level, our mapping scheme automatically assigns the same leading processor for every *parent-left child* pair. Needless to say, only one submatrix will actually be relocated after each submatrix merging operation.

3. The analysis of the model problem can be greatly simplified by examining the work associated with a branch of the row merge tree, which corresponds to the critical path of the parallel algorithm. This point will be made more clear in the following section on complexity analysis.

Complexity Analysis

The following analysis is for a k-by-k grid model problem on a hypercube multiprocessor of dimension d. For convenience, we shall assume $k = 2^{\ell} - 1$, where $\ell > 0$. Letting p denote the total number of node processors on the machine, we have $p = 2^d$. Since all of the p processors cooperate to perform the last task and the work involves merging two full k-by-k upper triangular matrices, we shall assume $k \gg p$ from now on.

We show in Figure 4.33 how to generate a nested dissection ordering by recursively defining separators on a 7-by-7 grid. The elimination tree corresponding to the grid in Figure 4.33 was displayed in Figure 4.34. Recall that the (reduced) row merge tree is transformed from the elimination tree by amalgamating the vertices along a chain into a single vertex. Therefore, there is a one-to-one mapping between the separators defined on the grid and the vertices in the row merge tree. Each vertex in the row merge tree carries the smallest label among the vertices in the corresponding separator. We give in Figure 4.35 the row merge tree for the 7-by-7 grid shown in Figure 4.33.

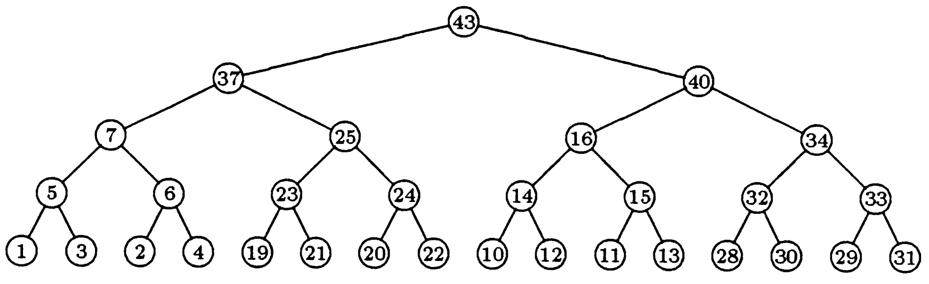


Figure 4.35: The (reduced) row merge tree associated with the 7-by-7 grid shown in Figure 4.33.

Now, with $p = 2^d$ processors available, the subtrees rooted at separators S_i^d, $1 \leq i \leq 2^d$, will each be assigned to one processor, and the separators one level above will each be assigned to a loop of two processors, and so on. The last separator, S_1^0 which is the root of the row merge tree, will be assigned a loop of p processors. Therefore, if we let $T\left(S_i^j, \frac{p}{2^j}\right)$ denote the time (computation and communication) required by the *parallel* submatrix merging operation associated with one task node, and let $T_s\left(S_i^j\right)$ denote the time for processing the subtree rooted at separator S_i^j by the serial row merging scheme, the total time required by the parallel row merging scheme to factor the coefficient matrix associated with the k-by-k grid can be expressed as

$$T_{k\times k}(k,p) = T_s\left(S_i^d\right) + \sum_{j=0}^{d-1} T\left(S_i^j, \frac{p}{2^j}\right),$$

where each S_i^j refers to one particular j^{th} level separator which is located on the *highest-cost* path of the row merge tree. Thus the values of i's may not be the same for all S_i^j. We shall explain shortly how to determine the highest-cost path. If we let $\left|S_i^j\right|$ denote the number of grid vertices contained in the separator (or the size of the separator), and we assume that $k = 2^\ell - 1$, $\ell > 0$, then we have

$$\left|S_1^0\right| = k,$$

$$\left|S_i^j\right| = \frac{\left|S_i^{(j-1)}\right| - 1}{2}, \; \textit{if } j \textit{ is odd} \, ,$$

and

$$\left|S_i^j\right| = \left|S_i^{(j-1)}\right|, \; \textit{if } j \textit{ is even}.$$

Note that the subgrids produced by the separators may have one more grid line along one or more sides. Such n-by-n subgrids are termed *bordered* subgrids in [33]. Given in Figure 4.36 is a subgrid bordered by separators on its two sides.

○	○	○	●
○	○	○	●
○	○	○	●
●	●	●	●

21	25	22	S_1^0
23	26	24	S_1^0
19	27	20	S_1^0
S_1^1	S_1^1	S_1^1	S_1^0

Figure 4.36: A bordered subgrid from the 7-by-7 grid in Figure 4.33.

Now if we let $\rho(n, i, q)$ be the cost (computation and communication) of factoring on q processors the coefficient matrix associated with an n-by-n subgrid which is bordered along i sides, then we have

$$\rho(n, j, q) > \rho(n, i, q), \quad \text{for every } j > i.$$

Therefore, the highest-cost path (which is not unique) can be defined by the separators in Figure 4.37. Applying this to the 7×7 grid in Figure 4.33, the corresponding branch on its row merge tree is identified by the path labelled in Figure 4.38. We can then derive the total cost of the parallel algorithm by setting up the recurrence equations for

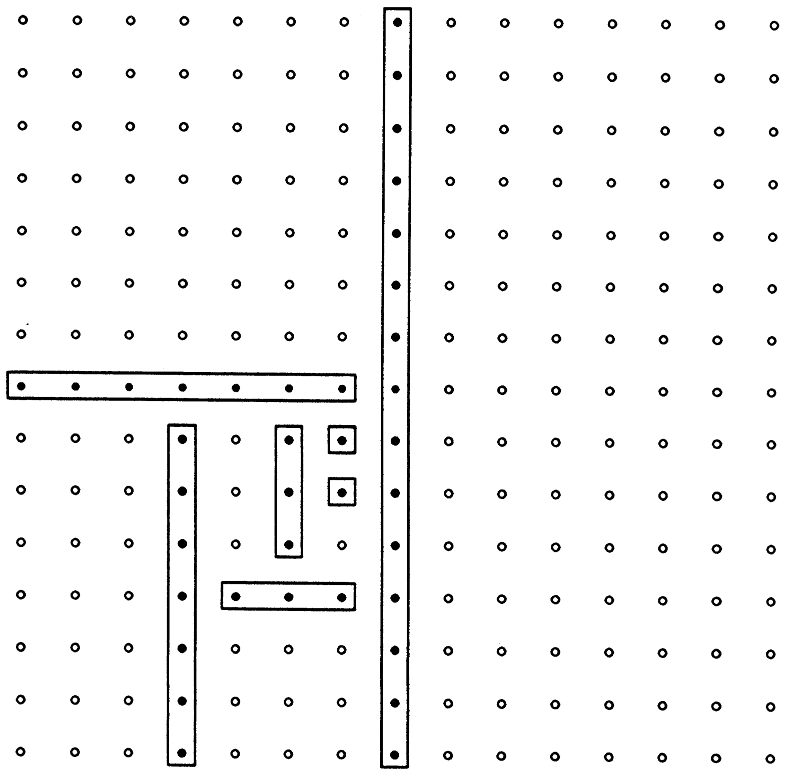


Figure 4.37: Separators along the highest-cost path of a 15-by-15 grid.

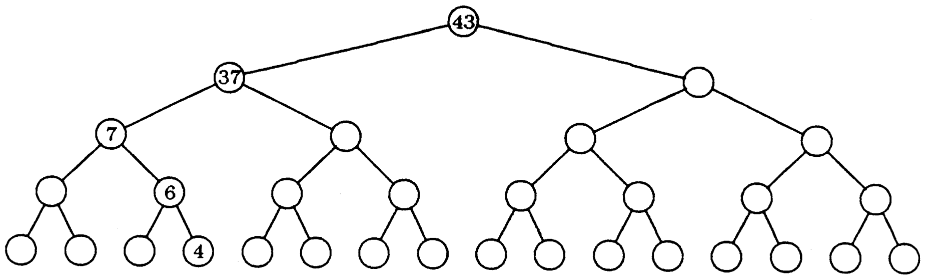


Figure 4.38: An example of the highest-cost branch on a (reduced) row merge tree.

the subproblems along the highest-cost path. We first define some cost functions which are needed in the recurrence equations.

Definition 3 *The function $\Lambda(u, v, t, q)$ is defined to be the computation and communication cost (in terms of multiplicative operations) of merging (via the parallel pairwise Givens scheme on q processors) two essentially full upper triangular submatrices of dimension u and v respectively, where t columns in these matrices have common subscripts.*

Definition 4 *$c(u, v, t)$ is defined to be $\Lambda(u, v, t, 1)$.*

Definition 5 *$\theta(k, i)$ is defined to be the multiplicative operations required to factor a coefficient matrix associated with a k-by-k grid bordered along i sides by the serial row merging scheme.*

We first show how to apply these cost functions recursively to the model problem using a 7-by-7 grid as an example. The nested dissection ordering of the grid was given in Figure 4.33 and a highest-cost branch in the corresponding row merge tree was given in Figure 4.38. As noted in Figure 4.38, the highest-cost path consists of vertices 43, 37, 7, 6 and 4. To apply the cost functions to the associated problems, let us recall the following.

1. There is a one-to-one mapping between the rows of the factor R and the vertices of the elimination tree. In particular, the s^{th} row of R is associated with the vertex $v_i = s$.
2. The nested dissection ordering dictates that the vertices contained in each separator are numbered sequentially.
3. Each separator S_j^i corresponds to a chain of length $|S_j^i|$ in the elimination tree.
4. Since the chain of vertices in the elimination tree is amalgamated into one single vertex in the (reduced) row merge tree, the observations 1 to 3 imply that associated with each vertex in the row merge tree is a collection of consecutive rows in R. Therefore,

the nonzero structure of these rows combined gives the column subscript set associated with the representative vertex of the chain, or vice versa.

Now, referring to the row merge tree in Figure 4.38, we see that the task associated with the root vertex 43 is the merging of two submatrices resulting from solving the subproblems induced by the separator S_1^0. Each subproblem is associated with a 7-by-3 subgrid bordered on one side as shown in Figure 4.39. The column subscript set of each submatrix corresponds to the vertices contained in the border. Because these two subgrids share a common border consisting of vertices $\{43, 44, \cdots, 49\}$, the task associated with vertex 43 is the merging of two full 7×7 upper triangular matrices with identical column subscript set of $\{43, 44, \cdots, 49\}$. Therefore, the cost function associated with the root vertex 43 is

21	25	22	●	●	30	34	31	o	o	o	43	o	o	o
23	26	24	●	●	32	35	33	o	o	o	44	o	o	o
19	27	20	●	●	28	36	29	o	o	o	45	o	o	o
37	38	39	●	●	42	41	40	o	o	o	46	o	o	o
3	7	4	●	●	12	16	13	o	o	o	47	o	o	o
5	8	6	●	●	14	17	15	o	o	o	48	o	o	o
1	9	2	●	●	10	18	11	o	o	o	49	o	o	o

Figure 4.39: Subproblems associated with subgrids.

given by $\Lambda(k, k, k, p)$, where $k = 7 = |S_1^0|$, p is the total number of processors available on the hypercube. The next vertex along the highest-cost path is vertex 37. Referring to the grid, vertex 37 is the representative of the separator S_1^1 which is consisted of vertices 37, 38 and 39. The separator S_1^1 further divides the 7-by-3 subgrid into two 3-by-3 subgrids. Each of the 3-by-3 subgrid is bordered along two sides. The submatrices produced from solving each of the subproblems associated with the two 3-by-3 subgrids have each a column subscript set containing the vertices in their respective borders. Therefore, the task associated with vertex 37 is the merging of two submatrices with column subscript sets of $\{37, 38, 39, 43, 44, 45, 46\}$ and $\{37, 38, 39, 46, 47, 48, 49\}$. These two column subscript sets can be identified from the grid as shown in Figure 4.40. The subscripts in the common set

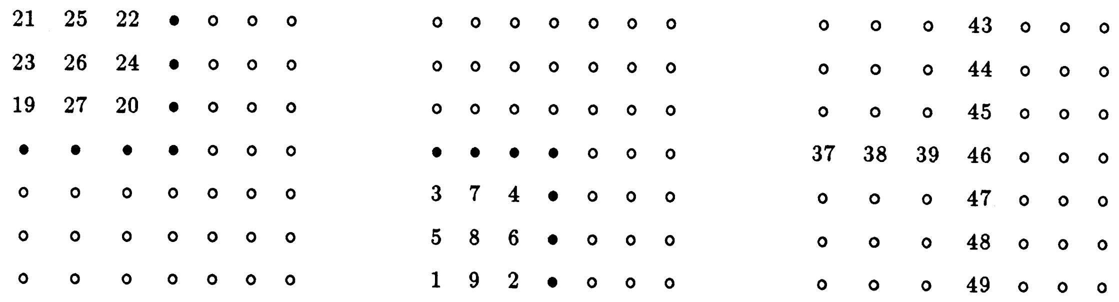


Figure 4.40: Subproblems and column subscript sets for merging at vertex 37.

are $\{37, 38, 39, 46\}$. In terms of the grid size k, the common subscript set is of size $(k/2)$, and each of the two column subscript sets is of size k. Therefore, the submatrices participating in the merging are each at most a $k \times k$ essentially full upper triangular matrix. According to our mapping strategy, $(p/2)$ processors will be allocated to perform this task. The cost for the merging operation associated with vertex 37 is thus $\Lambda\left(k, k, \frac{k}{2}, \frac{p}{2}\right)$. A task of exactly the same cost is associated with vertex 40 and is performed by the other set of $(p/2)$ processors in parallel.

The cost analysis given above can be applied to each of the four subproblems defined by the four subgrids induced by the first two levels of separators. We display the four subgrids in Figure 4.41. Note that each one of them is a square grid of dimension $(k/2)$ and is bordered on two sides. According to our mapping strategy, the four associated subproblems are solved in parallel, each employing $(p/4)$ processors. The cost for solving any one of them is therefore $\rho\left(\frac{k}{2}, 2, \frac{p}{4}\right)$.

Applying the analysis above recursively, we can now set up the following recurrence

21	25	22	●	30	34	31
23	26	24	●	32	35	33
19	27	20	●	28	36	29
●	●	●	●	●	●	●
3	7	4	●	12	16	13
5	8	6	●	14	17	15
1	9	2	●	10	18	11

Figure 4.41: The four subgrids induced by the separators.

equations for the subproblems along the highest-cost path, which determines the total cost of the parallel row merging scheme. We have

$$\rho(k,0,p) \leq \rho\left(\frac{k}{2},2,\frac{p}{4}\right) + \Lambda(k,k,k,p) + \Lambda\left(k,k,\frac{k}{2},\frac{p}{2}\right)\ , \tag{4.2}$$

$$\rho(k,2,p) \leq \rho\left(\frac{k}{2},4,\frac{p}{4}\right) + \Lambda\left(\frac{5k}{2},\frac{3k}{2},k,p\right) + \Lambda\left(2k,\frac{3k}{2},\frac{k}{2},\frac{p}{2}\right)\ , \tag{4.3}$$

$$\rho(k,4,p) \leq \rho\left(\frac{k}{2},4,\frac{p}{4}\right) + \Lambda(3k,3k,k,p) + \Lambda\left(2k,2k,\frac{k}{2},\frac{p}{2}\right)\ , \tag{4.4}$$

and

$$\begin{aligned} \rho\left(\frac{k}{2^{d/2}},4,1\right) &\leq \theta\left(\frac{k}{2^{d/2}},4\right) \\ &= \theta\left(\frac{k}{\sqrt{p}},4\right)\ , \end{aligned} \tag{4.5}$$

where d is the dimension of the hypercube machine, and is assumed to be an *even* number here so that each processor will be assigned a square subgrid. This assumption is made to facilitate our recurrence analysis, although the algorithm we propose is valid for any value of d. Since the cost functions $\Lambda(u,v,t,q)$ and $\theta\left(\frac{k}{\sqrt{p}},4\right)$ yield upper bounds, we have inequalities in the equations above. In what follows, we shall first show how to solve for $\theta(k,i)$ to obtain the serial cost $\theta(k,0)$ and evaluate $\theta\left(\frac{k}{\sqrt{p}},4\right)$ which is needed in (4.5). We next show that the cost function $\Lambda(u, v, t, q)$ is the summation of the arithmetic cost, the start-up time cost, the communication overhead cost and the cost for data distribution and collection for each submatrix operation. To solve for the total arithmetic cost, $\sigma(k, 0, p)$, the total start-up time, $\psi_1(k, 0, p)$, the total communication overhead, $\psi_2(k, 0, p)$ and the total cost for data distribution and collection, $\psi_3(k, 0, p)$, we replace the function $\Lambda(u, v, t, q)$ in (4.2)–(4.4) by each component cost function. Finally, we obtain an upper bound for $\rho(k, 0, p)$ from

$$\rho(k, 0, p) < \sigma(k, 0, p) + \psi_1(k, 0, p) + \psi_2(k, 0, p) + \psi_3(k, 0, p) .$$

•

To obtain $\theta(k, i)$, we need the formula for $c(u, v, t)$ which was derived in [62] and we cite it below in Lemma 4.3. Note that the cost function $c(w, w, w)$ we use in Lemma 4.3 measures the number of multiplicative floating point operations for merging two full $w \times w$ upper triangular matrices using either Method A (standard Givens) or Method B (pairwise Givens), whereas Liu's formula measures the number of elements being modified. Lemma 4.2 ensures that the cost functions in Lemma 4.3 yield upper bounds.

Lemma 4.2 [62] *The cost of merging two essentially full upper triangular submatrices $U = TZ[u, S_1]$ and $V = TZ[v, S_2]$ using Givens rotations, where $|S_1 \cap S_2| = t$, is bounded by the cost of merging U and V when their leading t columns have common subscripts.*

□

Lemma 4.3 [62] *Consider merging two essentially full upper triangular submatrices $U = TZ[u, S_1]$ and $V = TZ[v, S_2]$ using Givens rotations, where the first t columns of U and V have common subscripts. Let $g(u, v, t)$ denote the total number of rotations. Then*

$$g(u, v, t) = t(w - t) + \frac{t(t+1)}{2} ,$$

where $w = u+v-t$. Let $c(u,v,t)$ denote the total arithmetic cost (in terms of multiplicative operations). Then

$$c(u,v,t) = c(w,w,w) - c(x,x,x),$$

where $w = u+v-t$, $x = w-t$, and $c(k,k,k) = \frac{2}{3}k^3 + 2k^2 + \frac{4}{3}k$ for $k = w$ and $k = x$.

□

The recurrence equations for computing $\theta(k,i)$, the total multiplicative operations required to factor a coefficient matrix associated with a k-by-k grid bordered along i sides, were set up in [62] as given by equations (4.6)–(4.9).

$$\begin{aligned}
\theta(k,0) &= 4\theta\left(\frac{k}{2},2\right) + 2c\left(k,k,\frac{k}{2}\right) + c(k,k,k)\,, && (4.6)\\
\theta(k,2) &= \theta\left(\frac{k}{2},2\right) + 2\theta\left(\frac{k}{2},3\right) + \theta\left(\frac{k}{2},4\right) + c\left(\frac{3k}{2},k,\frac{k}{2}\right) + c\left(2k,\frac{3k}{2},\frac{k}{2}\right)\\
&\quad + c\left(\frac{5k}{2},\frac{3k}{2},k\right)\,, && (4.7)\\
\theta(k,3) &= 2\theta\left(\frac{k}{2},3\right) + 2\theta\left(\frac{k}{2},4\right) + 2c\left(2k,\frac{3k}{2},\frac{k}{2}\right) + c\left(\frac{5k}{2},\frac{5k}{2},k\right)\,, && (4.8)\\
\theta(k,4) &= 4\theta\left(\frac{k}{2},4\right) + 2c\left(2k,2k,\frac{k}{2}\right) + c(3k,3k,k)\,. && (4.9)
\end{aligned}$$

The solutions we obtain are given in Lemma 4.4.

Lemma 4.4

$$\begin{aligned}
\theta(k,4) &= \frac{371}{3}k^3 + 31k^2\log_2 k - 121k^2 - \frac{8}{3}k\,, && (4.10)\\
\theta(k,3) &= \frac{283}{3}k^3 + 31k^2\log_2 k - 133k^2 + \frac{116}{3}k\,, && (4.11)\\
\theta(k,2) &= \frac{499}{7}k^3 + \frac{155}{3}k^2\log_2 k - 145k^2 + 80k - \frac{44}{7}\,, && (4.12)\\
\theta(k,0) &= \frac{829}{21}k^3 + \frac{155}{3}k^2\log_2 k - \frac{569}{3}k^2 + \frac{488}{3}k - \frac{176}{7}\,. && (4.13)
\end{aligned}$$

Proof: The recurrence equations given by (4.6)–(4.9) are solved in the order $\theta(k,4)$, $\theta(k,3)$, $\theta(k,2)$ and $\theta(k,0)$. To solve for $\theta(k,4)$, we expand (4.9) as below so that $\theta(k,4)$ is explicitly expressed in terms of the known cost function $c(u,v,t)$.

$$\begin{aligned}\theta(k,4) &= 4\theta\left(\frac{k}{2},4\right)+2c\left(2k,2k,\frac{k}{2}\right)+c(3k,3k,k)\\ &= 2\sum_{i=1}^{\log_2 k}4^{i-1}c\left(\frac{2k}{2^{i-1}},\frac{2k}{2^{i-1}},\frac{k}{2^i}\right)+\sum_{i=0}^{\log_2 k-1}4^i c\left(\frac{3k}{2^i},\frac{3k}{2^i},\frac{k}{2^i}\right)\ .\end{aligned}$$

The above expression is further simplied using MAPLE [7] to obtain $\theta(k,4)$ in (4.10).

To solve for $\theta(k,3)$, we first replace $\theta\left(\frac{k}{2},4\right)$ in (4.8) by an explicit expression obtained from the solution for $\theta(k,4)$ in (4.10). The resulting equation involves $\theta(k,3)$ on the left hand side and $\theta\left(\frac{k}{2},3\right)$ on the right hand side. We can now proceed to solve for $\theta(k,3)$ in exactly the same way as we did above in solving for $\theta(k,4)$. $\theta(k,2)$ and $\theta(k,0)$ are subsequently solved in a similar manner. □

We derive next the arithmetic cost of solving the k-by-k grid model problem on a hypercube with p processors. Recall that in setting up the recurrence equations for the total cost $\rho(k,0,p)$ we employ the cost function $\Lambda(u,v,t,q)$, which includes both the *arithmetic* and *communication* cost of solving the associated problem. To obtain an upper bound of the arithmetic cost $\sigma(k,0,p)$ for the parallel algorithm, we have to solve the same set of recurrence equations except for replacing $\Lambda(u,v,t,q)$ by a proper cost function which computes the arithmetic cost alone for the associated problem. In Lemma 4.6 we show that the cost function to replace $\Lambda(u,v,t,q)$ in equations (4.2)–(4.4) is given by $C(u+v-t,t,q)$. To derive the latter and the communication cost, we need the results in the following lemma.

Lemma 4.5 *Consider merging a k-by-k full upper triangular matrix and a t-by-k ($t<k$) full upper trapezoidal matrix using a loop of p ($p<t$) processors via the pairwise Givens scheme. The total number of parallel steps (i rotations are performed by i processors concurrently per parallel step, where $1\le i\le p$) is given by*

$$G(k,t,p)=\frac{t}{2p}(2k-t+p)\ ,$$

and the arithmetic cost measured by the number of multiplicative operations is given by

$$\begin{aligned} C(k,t,p) &= 2\left(\frac{k^2t}{p} + p^2 - p - \frac{t^2}{p} + t - \frac{kt^2}{p}\right) \\ &\quad + 4\frac{kt}{p} + \frac{2}{3}\frac{t^3}{p} - \frac{2}{3}pt\,. \end{aligned}$$

Proof: For simplicity in our derivation, we shall assume that t and k are each an integral multiple of p. Using the wrap mapping scheme, each processor will have been allocated (t/p) rows and (k/p) rows from each matrix respectively. Since all of the p processors will start merging their first pair of rows simultaneously and the p^{th} processor will be kept busy at all steps until it last merges the last pair of k^{th} rows, the total number of parallel steps is equal to the number of rotations the p^{th} processor will perform, which is computed by

$$\begin{aligned} G(k,t,p) &= p\sum_{i=1}^{t/p} i + t\frac{(k-t)}{p} \\ &= \frac{t}{2p}(2k - t + p)\,. \end{aligned}$$

To compute the arithmetic cost of the parallel algorithm, we need to account for the maximum number of multiplicative operations performed at each step. Using the wrap mapping scheme and assuming that t and k are each an integral multiple of p, the maximum number of operations are performed by the first processor which is assigned row 1, row $(p+1)$, $\cdots$, and row $(k-p+1)$ before it has exhausted its data. After that the remaining $(p-1)$ processors will each update their last row in the remaining $(p-1)$ steps subsequently. Therefore, the total number of multiplicative operations is given by

$$\begin{aligned} C(k,t,p) &= 4\sum_{i=0}^{\frac{t}{p}-1}(1+ip)(k-ip) + 4\sum_{i=0}^{\frac{k-t}{p}-1} t(k-t-ip) + 4\sum_{i=1}^{p-1} i \\ &= 2\left(\frac{k^2t}{p} + p^2 - p - \frac{t^2}{p} + t - \frac{kt^2}{p}\right) \\ &\quad + 4\frac{kt}{p} + \frac{2}{3}\frac{t^3}{p} - \frac{2}{3}pt\,. \end{aligned}$$

□

Lemma 4.6 *We consider merging two essentially full upper triangular submatrices $U = TZ[u, S_1]$ and $V = TZ[v, S_2]$, where $|S_1 \cap S_2| = t$, using a loop of q ($q < t$) processors via the pairwise Givens scheme. Let $w = (u + v - t)$. The total number of parallel steps is given by*

$$G(w,t,q) \leq \frac{t}{2q}(2w - t + q) ,$$

and the total number of multiplicative operations is given by

$$\begin{aligned} C(w,t,q) \ &\leq \ 2\left(\frac{w^2t}{q} + q^2 - q - \frac{t^2}{q} + t - \frac{wt^2}{q}\right) \\ &\quad + 4\frac{wt}{q} + \frac{2}{3}\frac{t^3}{q} - \frac{2}{3}qt . \end{aligned}$$

Proof: To obtain an upper bound of the arithmetic cost, we assume that the columns with subscripts in common are the leading t columns. Our proof employs the following observation. Consider, for example, $U = TZ[6, \{1,2,3,4,5,6\}]$ and $V = TZ[5, \{1,2,7,8,9\}]$ in Figure 4.42. First, note that the merging of U and V in Figure 4.43 is equivalent to merging $\tilde{U}$ and $\tilde{V}$ in Figure 4.44. Second, the number of parallel steps and the arithmetic cost of merging $\tilde{U}$ and $\tilde{V}$ is bounded by those of merging a full $w \times w$ ($w = u + v - t = 9$) upper triangular matrix $\hat{U}$ and a $t \times w$ full upper trapezoidal matrix $\hat{V}$ in Figure 4.45. The total number of parallel steps and the multiplicative operations for merging $\hat{U}$ and $\hat{V}$ can then be obtained from $G(w,t,q)$ and $C(w,t,q)$ defined in Lemma 4.5. This proves the lemma. □

$$\left(\begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
u & u & u & u & u & u & & & \\
 & u & u & u & u & u & & & \\
 & & u & u & u & u & & & \\
 & & & u & u & u & & & \\
 & & & & u & u & & & \\
 & & & & & u & & &
\end{array}\right)
\qquad
\left(\begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
v & v & & & & & v & v & v \\
 & v & & & & & v & v & v \\
 & & & & & & v & v & v \\
 & & & & & & & v & v \\
 & & & & & & & & v \\
 & & & & & & & &
\end{array}\right)$$

Figure 4.42: The submatrices $U = TZ[6,\{1,2,3,4,5,6\}]$ and $V = TZ[5,\{1,2,7,8,9\}]$.

$$\left(\begin{array}{ccccccccc}
u & u & u & u & u & u & & & \\
 & u & u & u & u & u & & & \\
 & & u & u & u & u & & & \\
 & & & u & u & u & & & \\
 & & & & u & u & & & \\
 & & & & & u & & & \\
- & - & - & - & - & - & - & - & - \\
v & v & & & & & v & v & v \\
 & v & & & & & v & v & v \\
 & & & & & & v & v & v \\
 & & & & & & & v & v \\
 & & & & & & & & v
\end{array}\right)
\longrightarrow
\left(\begin{array}{ccccccccc}
\star & \star & \star & \star & \star & \star & \star & \star & \star \\
 & \star & \star & \star & \star & \star & \star & \star & \star \\
 & & \star & \star & \star & \star & \star & \star & \star \\
 & & & \star & \star & \star & \star & \star & \star \\
 & & & & \star & \star & \star & \star & \star \\
 & & & & & \star & \star & \star & \star \\
 & & & & & & \star & \star & \star \\
 & & & & & & & \star & \star \\
 & & & & & & & & \star \\
 & & & & & & & & \\
 & & & & & & & & \\
 & & & & & & & &
\end{array}\right)$$

Figure 4.43: Merging $U = TZ[6,\{1,2, 3,4,5,6\}]$ and $V = TZ[5,\{1,2,7,8,9\}]$.

$$\left(\begin{array}{ccccccccc}
u & u & u & u & u & u & & & \\
 & u & u & u & u & u & & & \\
 & & u & u & u & u & & & \\
 & & & u & u & u & & & \\
 & & & & u & u & & & \\
 & & & & & u & & & \\
 & & & & & & v & v & v \\
 & & & & & & & v & v \\
 & & & & & & & & v \\
- & - & - & - & - & - & - & - & - \\
v & v & & & & & v & v & v \\
 & v & & & & & v & v & v
\end{array}\right)
\longrightarrow
\left(\begin{array}{ccccccccc}
\star & \star & \star & \star & \star & \star & \star & \star & \star \\
 & \star & \star & \star & \star & \star & \star & \star & \star \\
 & & \star & \star & \star & \star & \star & \star & \star \\
 & & & \star & \star & \star & \star & \star & \star \\
 & & & & \star & \star & \star & \star & \star \\
 & & & & & \star & \star & \star & \star \\
 & & & & & & \star & \star & \star \\
 & & & & & & & \star & \star \\
 & & & & & & & & \star \\
 & & & & & & & & \\
 & & & & & & & & \\
 & & & & & & & &
\end{array}\right)$$

Figure 4.44: Merging $\tilde{U}$ and $\tilde{V}$.

$$\begin{pmatrix}
\times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & \times & \times & \times & \times & \times & \times & \times & \times \\
 & & \times & \times & \times & \times & \times & \times & \times \\
 & & & \times & \times & \times & \times & \times & \times \\
 & & & & \times & \times & \times & \times & \times \\
 & & & & & \times & \times & \times & \times \\
 & & & & & & \times & \times & \times \\
 & & & & & & & \times & \times \\
 & & & & & & & & \times \\
- & - & - & - & - & - & - & - & - \\
\times & \times & \times & \times & \times & \times & \times & \times & \times \\
 & \times & \times & \times & \times & \times & \times & \times & \times
\end{pmatrix}
\longrightarrow
\begin{pmatrix}
\star & \star & \star & \star & \star & \star & \star & \star & \star \\
 & \star & \star & \star & \star & \star & \star & \star & \star \\
 & & \star & \star & \star & \star & \star & \star & \star \\
 & & & \star & \star & \star & \star & \star & \star \\
 & & & & \star & \star & \star & \star & \star \\
 & & & & & \star & \star & \star & \star \\
 & & & & & & \star & \star & \star \\
 & & & & & & & \star & \star \\
 & & & & & & & & \star \\
 & & & & & & & & \\
 & & & & & & & & \\
 & & & & & & & &
\end{pmatrix}$$

Figure 4.45: Merging $\hat{U}$ and $\hat{V}$.

We now apply the cost function above to the k-by-k grid model problem on a hypercube with p processors. An upper bound of the arithmetic cost for the parallel row merging scheme, denoted by $\sigma(k, 0, p)$, can be obtained by solving the following recurrence equations, which are modified from Equations (4.2)–(4.5) by replacing $\Lambda(u, v, t, q)$ by $C(u + v - t, t, q)$.

$$\sigma(k,0,p) \leq \sigma\left(\frac{k}{2},2,\frac{p}{4}\right) + C(k,k,p) + C\left(\frac{3k}{2},\frac{k}{2},\frac{p}{2}\right) , \tag{4.14}$$

$$\sigma(k,2,p) \leq \sigma\left(\frac{k}{2},4,\frac{p}{4}\right) + C\left(3k,k,p\right) + C\left(3k,\frac{k}{2},\frac{p}{2}\right) , \tag{4.15}$$

$$\sigma(k,4,p) \leq \sigma\left(\frac{k}{2},4,\frac{p}{4}\right) + C(5k,k,p) + C\left(\frac{7k}{2},\frac{k}{2},\frac{p}{2}\right) , \tag{4.16}$$

$$\sigma\left(\frac{k}{\sqrt{p}},4,1\right) \leq \theta\left(\frac{k}{\sqrt{p}},4\right) . \tag{4.17}$$

Solving Equations (4.14) – (4.17), we obtain the following lemma.

Lemma 4.7 *The total arithmetic cost for applying the parallel row merging scheme to the*

k-by-k grid model problem on a hypercube having p processors is given by

$$\sigma(k,0,p) \leq \frac{146}{3}\frac{k^3}{p} + 31\frac{k^2\log_2 k}{p} - 155\frac{k^2}{p} - \frac{20}{21}kp + 6k - \frac{54}{7}\frac{k}{\sqrt{p}} + \frac{8}{3}p^2 - 4p + \frac{4}{3}\,.$$

□

We now proceed to analyze the communication cost for the parallel row merging scheme when applied to the k-by-k grid model problem on a hypercube having p processors. We distinguish three kinds of communication costs. They are start-up time, the communication overhead during each submatrix merging operation, and the time for data distribution and collection before and after each submatrix merging operation. The upper bounds we shall provide are $\psi_1(k,0,p)$ for total start-up time, $\psi_2(k,0,p)$ for total communication overhead, and $\psi_3(k,0,p)$ for total cost of data distribution and collection.

To obtain $\psi_1(k,0,p)$, we solve the same set of recurrence equation as given by Equations (4.14)–(4.17) except for noting that the arithmetic cost function should be replaced by a proper start-up time cost function and that the communication cost is zero when the subproblem is solved by a single processor. The start-up time cost function is given in Lemma 4.8.

Lemma 4.8 *The start-up time incurred in merging two essentially full upper triangular submatrices $U = TZ[u,S_1]$ and $V = TZ[v,S_2]$, where $|S_1 \cap S_2| = t$, using a loop of q ($q < t$) processors via the pairwise Givens scheme is given by*

$$\mu(w,t,q) = \frac{t}{2q}(2w - t + q)\beta\ ,$$

where $w = u + v - t$, and β is the start-up time for sending a message.

Proof: The result above is immediate from Lemma 4.6 and that for each parallel step, we only need to account for the start-up time for one message because all processors will be sending one row to their right neighbours simultaneously. We thus have $\mu(w,t,q) = G(w,t,q)\beta$. □

The set of recurrence equations are modified from Equations (4.14)–(4.17) and are given below by Equations (4.18)–(4.21).

$$\psi_1(k,0,p) = \psi_1\left(\frac{k}{2},2,\frac{p}{4}\right) + \mu(k,k,p) + \mu\left(\frac{3k}{2},\frac{k}{2},\frac{p}{2}\right) , \tag{4.18}$$

$$\psi_1(k,2,p) = \psi_1\left(\frac{k}{2},4,\frac{p}{4}\right) + \mu(3k,k,p) + \mu\left(3k,\frac{k}{2},\frac{p}{2}\right) , \tag{4.19}$$

$$\psi_1(k,4,p) = \psi_1\left(\frac{k}{2},4,\frac{p}{4}\right) + \mu(5k,k,p) + \mu\left(\frac{7k}{2},\frac{k}{2},\frac{p}{2}\right) , \tag{4.20}$$

$$\psi_1\left(\frac{k}{\sqrt{p}},4,1\right) = 0 . \tag{4.21}$$

Solving Equations (4.18)–(4.21) we obtain $\psi_1(p,0,k)$ as given in Lemma 4.9.

Lemma 4.9 *The total start-up time incurred in applying the parallel row merging scheme to the k-by-k grid model problem on a hypercube having p processors is given by*

$$\psi_1(k,0,p) = \beta\left(\frac{31}{8}\frac{k^2\log_2 p}{p} - \frac{17}{2}\frac{k^2}{p} + \frac{3}{2}k - \frac{3}{2}\frac{k}{\sqrt{p}}\right) .$$

For the communication overhead involved in each submatrix merging operation, we can substitute ψ_1 by ψ_2, and $\mu(w,t,q)$ by $\phi(w,t,q)$ in (4.18) to (4.21), where $\phi(w,t,q)$ is the data transmission cost given in the following lemma.

Lemma 4.10 *Consider merging two essentially full upper triangular submatrices $U = TZ[u,S_1]$ and $V = TZ[v,S_2]$, where $|S_1 \cap S_2| = t$, using a loop of q ($q < t$) processors via the pairwise Givens scheme. Recall that each processor must send the reduced row to its neighbour after every row merging operation. The communication cost (excluding the start-up time) is given by*

$$\phi(w,t,q) \le \frac{\alpha}{4}C(w,t,q) ,$$

where $w = u+v-t$ and α is the ratio of the time for transmitting one floating-point number across one link to the time for one floating-point multiplicative operation.

Proof: For each parallel step, the communication overhead is the time for sending the longest row across one link, and the computation cost is to merge the pair of longest rows. Therefore, the communication overhead per parallel step is $(\alpha/4)$ times the computational cost, so is the total communication cost. $\square$

Solving Equations (4.17)– (4.21) after replacing every occurrence of $\mu(w,t,q)$ by $\phi(w,t,q)$, we obtain the total communication overhead cost as given in Lemma 4.11.

Lemma 4.11 *When applying the parallel row merging scheme to the k-by-k grid model problem on a hypercube having p processors, the total communication cost for data transmission during the submatrix merging operations (excluding start-up time) is given by*

$$\begin{aligned}\psi_2(k,0,p) \;&\leq\; \alpha\left(\frac{73}{6}\frac{k^3}{p} - \frac{371}{12}\frac{k^3}{p\sqrt{p}} + \frac{31}{8}\frac{k^2\log_2 p}{p} - \frac{17}{2}\frac{k^2}{p}\right.\\ &\quad\left. + \frac{53}{42}\frac{k}{\sqrt{p}} - \frac{5}{21}kp + \frac{3}{2}k + \frac{2}{3}p^2 - p + \frac{1}{3}\right) .\end{aligned}$$

$\square$

To compute the upper bound of communication cost for data distribution and collection, we derive in Lemma 4.12 an upper bound of the data distribution and collection cost for each submatrix merging operation.

Lemma 4.12 *Consider applying the parallel row merging scheme to the model problem on a hypercube having p processors. Associated with the merge of two essentially full upper triangular submatrices $U = TZ[u,S_1]$ and $V = TZ[v,S_2]$, where $|S_1 \cap S_2| = t$, an upper bound of the data collection and distribution cost is obtained by assuming a $w \times w$ full upper triangular matrix is redistributed and relocated before and after each submatrix merging operation, where $w = u + v - t$. Such cost incurred in each submatrix merging operation is given by*

$$\lambda(w)\log_2 p\ ,$$

where $\lambda(w) = w(w+1)\alpha$, $(\log_2 p)$ *is the dimension of the hypercube, and* α *is the ratio of the time for transmitting one floating-point number across one link to the time for one floating-point multiplicative operation.*

Proof: We noted earlier that at the end of each merging operation, only one submatrix needs to be relocated. The size of this submatrix is at most $w(w+1)/2$ and the path length is at most $\log_2 p$, resulting in the cost of $(w(w+1)\alpha\log_2 p)/2$. At the beginning of each step, the rows of the two submatrices must be distributed among the processors. Since each loop of processors form a subcube and we can pipeline the distribution, and each submatrix is of dimension $u \leq w$ or $v \leq w$, a reasonable upper bound is also $(w(w+1)\alpha\log_2 p)/2$. The cost of $\lambda(w)\log_2 p$ is obtained by summing up these two. □

The total cost for data distribution and collection can now be obtained by solving the set of recurrence equations (4.22) to (4.25), which are modified from Equations (4.18)–(4.21) by replacing every occurrence of $\mu(w,t,q)$ by $\lambda(w)\log_2 p$.

$$\psi_3(k,0,p) \leq \psi_3\left(\frac{k}{2},2,\frac{p}{4}\right) + \lambda(k)\log_2 p + \lambda\left(\frac{3k}{2}\right)\log_2 p\,, \tag{4.22}$$

$$\psi_3(k,2,p) \leq \psi_3\left(\frac{k}{2},4,\frac{p}{4}\right) + 2\lambda(3k)\log_2 p\,, \tag{4.23}$$

$$\psi_3(k,4,p) \leq \psi_3\left(\frac{k}{2},4,\frac{p}{4}\right) + \lambda(5k)\log_2 p + \lambda\left(\frac{7k}{2}\right)\log_2 p\,, \tag{4.24}$$

$$\psi_3\left(\frac{k}{\sqrt{p}},4,1\right) = 0\,. \tag{4.25}$$

The solution we obtain for $\psi_3(k,0,p)$ is given in Lemma 4.13.

Lemma 4.13 *When the parallel row merging scheme is applied to the k-by-k grid model problem on a hypercube having p processors, an upper bound of the cost for data distribution and collection is given by*

$$\psi_3(k,0,p) \leq \alpha\left(\frac{521}{48}k^2\log_2 p - \frac{149}{3}\frac{k^2\log_2 p}{p} + \frac{39}{4}k\log_2 p - 17\frac{k\log_2 p}{p}\right)\,.$$

□

Because $\sigma(k,0,p)$ and $(\psi_1(k,0,p) + \psi_2(k,0,p) + \psi_3(k,0,p))$ give the respective upper bounds for the total arithmetic cost and communication cost associated with the model problem, the total cost $\rho(k,0,p)$ can now be bounded by the sum of them. We give the result in Theorem 4.14.

Theorem 4.14 *An upper bound of the total cost for applying the parallel row merging scheme to a k-by-k grid model problem on a hypercube having p processors is given by*

$$\begin{aligned}\rho(k,0,p) \;<\; & \left(\frac{146}{3} + \frac{146}{12}\alpha - \frac{371}{12\sqrt{p}}\alpha\right)\frac{k^3}{p} + 31\frac{k^2\log_2 k}{p} + \frac{521}{48}k^2(\log_2 p)\alpha \\ & + \left(\frac{31}{8}\beta - \frac{1099}{24}\alpha\right)\frac{k^2\log_2 p}{p} - 155\frac{k^2}{p} - \frac{17}{2}\frac{k^2}{p}(\alpha+\beta) \\ & - \frac{20}{21}kp + O(k\log_2 p)\,.\end{aligned}$$

Proof: To obtain an upper bound of the total cost, we sum up the upper bounds of total arithmetic cost, total start-up time, total communication overhead and the total cost for data distribution and collection. The above result is thus immediate from the following inequality and Lemma 4.7, 4.9, 4.11 and 4.13.

$$\rho(k,0,p) < \sigma(k,0,p) + \psi_1(k,0,p) + \psi_2(k,0,p) + \psi_3(k,0,p)\,.$$

□

Observe in the analysis above that the coefficient of the $O(k^3/p)$ term in the parallel arithmetic cost $\sigma(k,0,p)$ is $(146/3)$, which is slightly bigger than the coefficient of $(829/21)$ of the $O(k^3)$ term of the serial arithmetic cost $\theta(k,0)$. They are not exactly the same because the tasks associated with the critical path are bigger than the tasks associated with other branches of the row merging tree. The upper bound we obtained in Theorem 4.14 for $\rho(k,0,p)$ indicates that the $\alpha O(k^3/p)$ communication cost of the proposed algorithm is of the same order of magnitude as the arithmetic cost. This is undesirable on a machine where the communication cost is not negligible compared to the arithmetic cost. Note that the function contributing to the $\alpha O(k^3/p)$ terms is $\psi_2(k,0,p)$ given in Lemma 4.11, which was computed by associating with each submatrix merging operation the communication overhead given by $\phi(w,t,q)$ in Lemma 4.10. In the next section we examine a generalized version of the parallel submatrix merging algorithm and indicate how it may reduce the communication cost.

4.3.6 Generalizing The Parallel Submatrix Merging Algorithm.

We first recall that the parallel pairwise Givens algorithm proposed in section 4.3.2 requires that the consecutive rows of both matrices are assigned to consecutive processors of the loop, with assignment "wrapping around" to processor 1 after a pair of rows is assigned to processor p. This mapping strategy can be viewed as a special case of a more general *block* wrap-mapping scheme, where the block size is equal to one. In Theorem 4.16 below we show that by choosing a particular block size the communication cost for merging two $k \times k$ full upper triangular matrices using a loop of p processors can be reduced from $O(k^3/p)$ to $O(pk^2)$, and that the leading term of the arithmetic cost remains unchanged. For any chosen block size $b \geq 1$, we first derive the arithmetic and communication cost functions in Lemma 4.15.

Lemma 4.15 *Consider merging two $k \times k$ full upper triangular matrices on a loop of p processors using the parallel pairwise block Givens scheme. For any chosen block size $b \geq 1$, the arithmetic cost $C_b(k,b,p)$ (measured by the total number of multiplicative operations) and the communication cost $\phi_b(k,b,p)$ (excluding the start-up time) are given by*

$$\begin{aligned} C_b(k,b,p) &= \frac{2}{3}\frac{k^3}{p} + 2\frac{k^2}{p} + 4b^2k - \frac{2}{3}b^2pk - 2bk - 4b^2\frac{k}{p} \\ &\quad + 4b\frac{k}{p} + 2p^2b^3 - 6b^3p + 4b^3 + 4pb^2 - 4b^2\ , \end{aligned}$$

and

$$\phi_b(k,b,p) = \frac{\alpha}{12}\left(2\frac{k^3}{bp} - 6k^2 + 3\frac{k^2}{p} + 3\frac{k^2}{bp} + 4bpk - 3bk\right.$$

$$- 3k - 6(b-1)\frac{k}{p} + 6b^2(p-2)^2 + 6b(p-2)\Big) .$$

Proof: Let b denote the block size. The block wrap mapping scheme will assign consecutive blocks, b rows per block, to consecutive processors in the loop, with assignment wrapping around to processor 1 after a pair of blocks is assigned to processor p. We shall use n_b to denote the number of block pairs, that is $n_b = \frac{k}{b}$. For convenience, we shall assume that k and n_b are each an integral multiple of p. By keeping track of the work performed by processor 1 before its data are exhausted as well as the work performed by the other $(p-1)$ processors in merging their last pair of submatrices, we obtain

$$\begin{aligned}
C_b(k,b,p) &= 4\sum_{j=0}^{\frac{n_b}{p}-1}(1+jp)\sum_{\ell=0}^{b-1}(k - jbp - \ell + k - jbp - b + 1 - \ell)\frac{b}{2} \\
&\quad + 4\sum_{j=0}^{p-2}\sum_{\ell=0}^{b-1}(b(p-1) - jb - \ell + b(p-1) - jb - b + 1 - \ell)\frac{b}{2} \\
&= \frac{2}{3}\frac{k^3}{p} + 2\frac{k^2}{p} + 4b^2k - \frac{2}{3}b^2pk - 2bk - 4b^2\frac{k}{p} \\
&\quad + 4b\frac{k}{p} + 2p^2b^3 - 6b^3p + 4b^3 + 4pb^2 - 4b^2 .
\end{aligned}$$

For $b \geq 1$, each processor will transmit the reduced submatrix to its neighbour after the leading b nonzeros in all of the b rows have been annihilated. By keeping track of the data transmitted by processor 1 as well as the data transmitted in the last $(p-2)$ parallel steps, we obtain

$$\begin{aligned}
\phi_b(k,b,p) &= \alpha\sum_{j=0}^{\frac{n_b}{p}-1}(1+jp)(k - (j+1)bp + k - (j+1)bp - b + 1)\frac{b}{2} \\
&\quad + \alpha\frac{b}{2}(p-2)(b(p-2)+1) \\
&= \frac{\alpha}{12}\left(2\frac{k^3}{bp} - 6k^2 + 3\frac{k^2}{p} + 3\frac{k^2}{bp} + 4bpk - 3bk\right. \\
&\quad \left. - 3k - 6(b-1)\frac{k}{p} + 6b^2(p-2)^2 + 6b(p-2)\right) ,
\end{aligned}$$

where α is the ratio of the time for transmitting one floating-point number across one link to the time for one floating-point multiplicative operation. $\square$

The results in Theorem 4.16 below can then be immediately obtained from Lemma 4.15.

Theorem 4.16 *Consider merging two k-by-k full upper triangular matrices on a loop of p processors using the parallel pairwise block Givens scheme. By choosing the block size $b = k/p^2$, the arithmetic cost $C_b(k,b,p)$ and $\phi_b(k,b,p)$ are given by*

$$\begin{aligned} C_b\left(k, \frac{k}{p^2}, p\right) &= \frac{2}{3}\frac{k^3}{p} + 2\frac{k^2}{p} - \frac{2}{3}\frac{k^3}{p^3} + 6\frac{k^3}{p^4} - 10\frac{k^3}{p^5} \\ &\quad + 4\frac{k^3}{p^6} - 2\frac{k^2}{p^2} + 8\frac{k^2}{p^3} - 4\frac{k^2}{p^4} \end{aligned}$$

and

$$\begin{aligned} \phi_b\left(k, \frac{k}{p^2}, p\right) &= \frac{\alpha}{12}\left(2pk^2 - 6k^2 + 3pk + 7\frac{k^2}{p} + 3\frac{k^2}{p^2}\right. \\ &\quad \left. -30\frac{k^2}{p^3} + 24\frac{k^2}{p^4} - 3k + 12\frac{k}{p} - 12\frac{k}{p^2}\right) . \end{aligned}$$

$\square$

Comparing $C_b(k,b,p)$ in Lemma 4.15 and the serial time given by

$$\frac{2}{3}k^3 + 2k^2 + \frac{4}{3}k ,$$

we see that the arithmetic cost of the generalized parallel pairwise Givens algorithm is optimal in its leading term. Note that the leading term of $C_b(k,b,p)$ is *independent* of the block size. Comparing $C_b(k,k/p^2,p)$ with $\phi_b(k,k/p^2,p)$, we see that the arithmetic cost dominates the communication cost when $k \gg p$. More specifically, the $O\left(k^3/p\right)$ arithmetic cost and $O\left(pk^2\right)$ communication cost imply that p should be chosen to be less than $\sqrt{k}$ in order to have the arithmetic cost dominate the communication cost. This implication is important because for the k-by-k grid model problem we have analyzed in this section, the last task on the critical path involves merging two $k \times k$ full upper triangular matrices using a loop of p processors.

The results in this section suggest that the communication cost of the parallel row merging scheme can be reduced by applying the generalized submatrix merging algorithm with appropriate block size to each task. The development of an enhanced parallel row merging scheme by incorporating this idea merits further research.

Chapter 5

Conclusions

5.1 A Summary of Contributions

In this thesis we have described three new parallel algorithms for reducing a rectangular matrix A to upper triangular form using orthogonal transformations. The first two algorithms we describe are for dense matrices on shared and local memory architectures respectively, and the third is for a class of sparse matrices on a local-memory multiprocessor.

In Chapter 2 we considered factoring a dense rectangular matrix on a shared-memory multiprocessor. We analyzed the synchronization cost, the work load distribution and the expected performance of the algorithm. Our analysis indicates that the proposed algorithm enjoys low synchronization cost. In particular, for a given $m \times n$ matrix A and a multiprocessor having p processing nodes, our analysis of the algorithm shows that this cost is $O\left(n^2/p\right)$ if $m/p \geq n$, and $O\left(mn/p^2\right)$ if $m/p < n$. Note that in the latter case, the synchronization cost is smaller than $O\left(n^2/p\right)$. Therefore, when $m \geq n$, the synchronization cost of the proposed algorithm is bounded by $O\left(n^2/p\right)$, which is *independent* of m. This is important for machines where synchronization cost is high, and when $m \gg n$. We have simulated high-synchronization cost in our experiments, and the timing results so obtained are in agreement with our analysis. The experiments also indicate that the algorithm is effective in balancing the load and producing high speed-up as predicted by our analysis.

In Chapter 3 we considered the problem of factoring a dense rectangular matrix on a hypercube multiprocessor. The proposed algorithm involves the embedding of a two-dimensional grid in the hypercube network, and our analysis of the algorithm determines how the aspect ratio of the embedded processor grid should be chosen in order to minimize the execution time or storage usage. The algorithm was implemented in FORTRAN and tested on an Intel iPSC hypercube with 64 processors. Our numerical experiments demonstrate the effect of the aspect ratio on the performance of the parallel algorithm and show that the execution time or storage requirement using the predicted aspect ratio is very close to the actual minimum for the test matrices.

Another feature of the algorithm proposed in Chapter 3 is that redundant computations are incorporated in a communication scheme which takes full advantage of the hypercube topology. With the proposed communication scheme the data are always exchanged between neighbouring processors and such exchanges can occur simultaneously on all channels. The latter feature is important in reducing traffic congestion in the network. It is expected that in future generations of hypercubes special hardware support may achieve a situation where sending a message to a processor several hops away may not take any longer than sending the message to a neighbour. However, the problem of traffic congestion will still exist. The communication scheme we proposed in Chapter 3 provides a solution to this problem.

The extensive experimental results presented in Chapter 3 also show that the proposed algorithm can be efficiently implemented and various enhancements can be easily incorporated to further reduce the execution time and storage requirement.

In Chapter 4 we considered the orthogonal decomposition of a class of large sparse matrices on a hypercube multiprocessor. The proposed algorithm offers a parallel implementation of the general row merging scheme for sparse Givens transformations developed by Liu [61]. The proposed parallel algorithm is novel in several aspects. First, we propose a new mapping strategy whose goal is to reduce the communication cost and balance the work load during the entire computing process. Second, we described a new sequential algorithm for merging two upper trapezoidal matrices (possibly of different dimensions), wherein the order of computation is different from the standard Givens scheme, and is more suitable for parallel implementation. Third, we show that the hypercube network can be employed as a multi-loop multiprocessor. The performance of the parallel algorithm applied to a model problem was analyzed and computation/communication complexity results were presented. Finally, we showed that the parallel submatrix merging algorithm can be viewed as a special case of a more general scheme and indicated how the generalized scheme may reduce the communication cost.

5.2 Further Work and Open Problems

Recall that when we applied Algorithm II in Chapter 3 to a dense square matrix, substantial saving in execution time and storage usage are obtained by embedding a two-dimensional grid in the hypercube network compared to employing the hypercube as a linear array. A natural question to ask is whether Algorithm II can be adapted to parallelize other numerical algorithms efficiently. In this section we give such an example by applying the ideas of Algorithm II to parallelize Gaussian elimination with pairwise pivoting on a hypercube multiprocessor. We briefly review the pairwise pivoting scheme and sketch how to adapt Algorithm II in Chapter 3 for this task.

The method of Gaussian elimination using triangularization by elementary stabilized matrices constructed by pairwise pivoting is analyzed by Sorensen in [83]. It is shown that a variant of this scheme which is suitable for implementation on a parallel computer is *numerically stable* although the error bound is larger than the one for the standard partial pivoting algorithm. The serial algorithm and its analysis are given in detail in [83]. For our purpose, it is sufficient to note that the variant we are considering can be understood as applying a 2×2 elementary matrix to each pair of rows in a similar fashion as applying Given's rotations. Recall that in the Given's scheme, we apply the rotation of the following form to a pair of rows to annihilate a leading nonzero element from one of the rows.

$$S = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} .$$

For Gaussian elimination with pairwise pivoting, this elementary 2×2 matrix will be of the following form

$$S = \begin{pmatrix} 1 & 0 \\ \gamma & 1 \end{pmatrix} P ,$$

where P is a 2×2 permutation. Therefore, to annihilate one element, only one row of data is modified. The serial arithmetic cost is therefore one half of the Given's scheme. If the work load is evenly distributed among the multiple processors, then the parallel arithmetic cost is also one half of the parallel Given's scheme. We can further improve the numerical stability without any cost by performing partial pivoting whenever parallelism can be maintained.

Following our description of Algorithm II in Chapter 3, we shall have each processor perform Gaussian elimination with "partial pivoting" in the IAP phase at each reduction step. After that all of the processors can cooperate to perform Gaussian elimination with "pairwise pivoting" in the CAP phase to eliminate the leading nonzeros in the local pivot rows. Note that with the wrap mapping a balanced work load distribution can be maintained throughout the entire elimination process as long as the k^{th} row of A is reduced to the k^{th} row of the upper triangular factor [8]. Therefore, explicit permutations during the CAP at the k^{th} reduction step are needed only when the pair of rows involve row k and row k is not chosen as the pivot row. Whenever this happens, our communication scheme ensures that both rows are present in the two processors involved. The explicit permutation can thus be done at no extra cost by carefully delaying the actual modification until the very last step. Another point worthy of noting is that there is *no* redundant computation involved simply because the rotation does not modify the row to be further exchanged! The analysis of the parallel scheme would be similar to the analysis of Algorithm II in Chapter 3.

In the remainder of this chapter we discuss several research problems which are related to our work and need further investigation.

1. It is well-known that the numerical factorization phase is the most time-consuming part in the process of solving linear equations or least squares problems on a sequential machine regardless of whether the matrix is *dense* or *sparse*. The algorithms proposed in this thesis have been concerned with parallelizing this phase. However, to solve the systems of linear equations or the least squares problems on a multiprocessor, we still require an efficient parallel algorithm to solve the resulting upper triangular system. In some cases such an algorithm can be easily adapted from some known scheme, but in some other cases an efficient triangular solver has not yet been developed.
   If A is dense and the target machine is a shared-memory multiprocessor, then the parallel QR factorization algorithm ensures that the rows of R be assigned to the processors according to a wrap mapping. The implementation of the parallel back substitution algorithm is straightforward and it is considered a solved problem [30].
   If A is dense and the target machine is a hypercube multiprocessor, we have two more cases to consider. Recall that our analysis and experimental results in Chapter 3 indicate that the p processors of the hypercube will be used as a p-by-1 grid or 1-by-p grid in case $m \gg n$ or $m \ll n$. Under these two extreme cases, our algorithm ensures that the *rows* (in the former case) or the *columns* (in the latter case) of factor R are wrap-mapped to the p processors. In either case there are a number of parallel algorithms available in the literature to solve the triangular system on the hypercube. The article by Heath and Romine [52] contains several new schemes and provides a comprehensive comparison of most triangular solvers available to date. Researchers in this area are still working on improving these schemes.

In the less extreme cases we have employed a two-dimensional grid in the factorization phase. Therefore, the rows and columns of the upper triangular factor R will have been wrap-mapped to the processors along the rows and columns of the grid respectively. In this case, it is not clear whether such a partitioning would allow an efficient triangular solver. This is a problem which merits further investigation.

When A is *sparse*, the parallel factorization algorithm we propose in Chapter 4 will have the rows of the upper triangular factor wrap-mapped on a loop of processors embedded in the hypercube. Although there is straightforward way to adapt currently available dense triangular solvers for the sparse case, the performance is far from being satisfactory [88]. Therefore, designing an efficient parallel solver for the resulting *sparse* triangular system remains a subject of further work.

2. All of the algorithms proposed in this thesis aim at achieving high performance on the target machine. Chapter 2 and Chapter 3 contain two very different algorithms designed for solving the same problem on two different classes of multiprocessors. These two very different designs support the common view that the design of parallel algorithms is architecture-dependent if *performance* is the ultimate goal. Since physical data partitioning is usually not a concern in designing parallel algorithms for shared-memory multiprocessors, the parallel algorithms in this class are obviously not portable to local-memory multiprocessors. On the other hand, it is relatively easy to simulate message-passing on a shared-memory machine. Therefore, the parallel algorithms designed for local-memory multiprocessors can be ported to shared-memory multiprocessors in a fairly simple way. Although we expect to lose some performance by porting the local-memory code to shared-memory machine by simulating message passing on the latter, the relative magnitude of the loss is unknown. Knowing that information would be useful in deciding whether the human effort involved in designing and implementing a separate scheme for the shared-memory machine is worthwhile.

We feel that the two algorithms in Chapter 2 and Chapter 3 are good candidates to investigate the issue of porting parallel algorithms across the two classes of multiprocessor architecture.

3. Throughout Chapter 4 we have assumed that the columns of the sparse matrix A have been appropriately ordered for reducing *fills* in the triangular factor R. For the class of matrices associated with the k-by-k grid model problem, it is well known that George's nested dissection ordering [27] attains the lower bound on the number of nonzero entries in R [33]. In Chapter 4 we explained in detail that the nested dissection ordering is also very suitable for parallel implementation of the general row merging scheme on the hypercube. Our complexity analysis provides upper bounds for the computation and communication cost of the parallel algorithm applied to the model problem. An enhanced parallel row merging scheme employing the generalized submatrix merging algorithm and an efficient implementation on a hypercube remain to be developed.

   For general sparse matrices, the structure of the row merge tree induced by a fill-reducing ordering may not be balanced and the size of each task may be drastically different. For the former problem, there exist heuristic algorithms which reorder the columns of A so that the induced row merge tree is better balanced while maintaining the same amount of fills in the factor R [63]. For the latter problem, our work in Chapter 3 shows that the hypercube connection allows us to embed multiple loops of different (even) sizes so that the allocation of processors to each task can be proportional to the amount of work involved. In this more general setting, a loop of processors assigned to an interior task node on the row merge tree may no longer share its leading processor with any one of its children nodes. Therefore, the relocation of submatrices between non-adjacent processors will occur more frequently. One crucial issue in an efficient implementation of the algorithm for general sparse matrices is the reduction of communication volume and traffic congestion.

Bibliography

[1] J. H. Argyris and O. E. Bronlund. The natural factor formulation of the stiffness matrix displacement method. *Comput. Meth. Appl. Mech. Engrg.*, 5:97–119, 1975.

[2] A. Bjorck. Solving linear least squares problems by Gram-Schmidt orthogonalization. *BIT*, 7:1–21, 1967.

[3] J. Blackmer, P. Kuekes, and G. Frank. A 200 MOPS systolic processor. In *Proc. SPIE, Vol. 298 (Real-Time Signal Processing IV)*, Society of Photo-Optical Instrumentation Engineers, 1982.

[4] K. Bromley, J. J. Symanski, J. M. Speiser, and H. J. Whitehouse. Systolic array processor developments. In H. T. Kung, R. F. Sproull, and Jr. G. L. Steeler, editors, *VISI Systems and Computations*, pages 273–284, Carnegie-Mellon University, Computer Science Press, October 1981.

[5] P. A. Businger and G. H. Golub. Linear least squares solutions by Householder transformations. *Numer. Math.*, 7:269–276, 1965.

[6] R. M. Chamberlain and M. J. D. Powell. *QR Factorization for Linear Least Squares Problems on the Hypercube.* Technical Report CCS 86/10, Dept. of Science and Technology, Chr. Michelsen Institute, Bergen, Norway, 1986.

[7] B. W. Char, K. O. Geddes, G. H. Gonnet, and S. M. Watt. *Maple User's Guide, 4th Edition.* WATCOM publications Limited, Waterloo, Ontario N2L 3X2, Canada, 1985.

[8] E. C. H. Chu and J. A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[9] M. Cosnard, J.-M. Muller, and Y. Robert. Parallel QR Decomposition of a Rectangular Matrix. *Numer. Math.*, 48:239–249, 1986.

[10] M. Cosnard and Y. Robert. Complexité de la factorisation QR en parallèle. *C.R. Acad. Sci.*, 297:549–552, 1983.

[11] G. J. Davis. *Column LU factorization with pivoting on a hypercube multiprocessor.* Technical Report 6219, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, 1985.

[12] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26:91–112, 1984.

[13] J. J. Dongarra, A. H. Sameh, and D. C. Sorensen. Implementation of some concurrent algorithms for matrix factorization. *Parallel Computing*, 3:25–34, 1985.

[14] I. S. Duff. Full matrix techniques in sparse Gaussian elimination. In G. A. Watson, editor, *Lecture Notes in Mathematics (912)*, Springer-Verlag, 1982.

[15] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3:193–204, 1986.

[16] I. S. Duff. Pivot selection and row ordering in Givens reduction on sparse matrices. *Computing*, 13:239–248, 1974.

[17] I. S. Duff and J. K. Reid. A comparison of some methods for the solution of sparse overdetermined systems of linear equations. *J. Inst. Maths. Appl.*, 17:267–280, 1976.

[18] L. Elden. *A Parallel QR Decomposition Algorithm.* Technical Report, Department of Scientific Computing, Uppsala University, and Department of Mathematics, Linkoping University, October 1987.

[19] M. Feilmeier. Parallel numerical algorithms. In D. Evans, editor, *Parallel Processing Systems*, pages 285–338, Cambridge University Press, 1982.

[20] M. J. Flynn. Very high speed computing systems. In *Proc. IEEE*, pages 1901–1909, 1966.

[21] D. B. Gannon and J. V. Rosendale. On the impact of communication complexity on the design of parallel numerical algorithms. *IEEE Trans. on Computers*, C-33:1180–1194, December 1984.

[22] G. A. Geist and M. T. Heath. *Parallel Cholesky factorization on a hypercube multiprocessor.* Technical Report ORNL-6211, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1985.

[23] W. M. Gentleman. Error analysis of QR decompositions by Givens transformations. *Linear Algebra and its Appl.*, 10:189–197, 1975.

[24] W. M. Gentleman. Least squares computations by Givens transformations without square roots. *J. Inst. Maths. Appl.*, 12:329–336, 1973.

[25] W. M. Gentleman. Row elimination for solving sparse linear systems and least squares problems. In G. A. Watson, editor, *Lecture Notes in Mathematics (506)*, pages 122–133, Springer-Veriag, 1975. (Proc. 1975 Dundee Conference on Numerical Analysis).

[26] W. M. Gentleman. Some complexity results for matrix computations on parallel processors. *J. Assoc. Comput. Mach.*, 25(1):112–115, 1978.

[27] J. A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[28] J. A. George and M. T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and its Appl.*, 34:69–83, 1980.

[29] J. A. George, M. T. Heath, and J. W-H. Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and its Appl.*, 77:165–187, 1986.

[30] J. A. George, M. T. Heath, J. W-H. Liu, and E. G-Y. Ng. *Solution of sparse positive definite systems on a shared memory multiprocessor.* Technical Report CS-86-10, Dept. of Computer Science, York University, 1986. (to appear in Internat. J. Parallel Programming).

[31] J. A. George, M. T. Heath, J. W-H. Liu, and E. G-Y. Ng. *Sparse Cholesky factorization on a local-memory multiprocessor.* Technical Report CS-86-02, Department of Computer Science, University of Waterloo, 1986. (to appear in Siam J. Sci. Stat. Comput.).

[32] J. A. George and J. W-H. Liu. Compact structural representation of sparse Cholesky, QR and LU factors. In R. Glowinski and J.-L. Lions, editors, *Computing Methods in Applied Sciences and Engineering, VII*, Elsevier Publishers B.V. (North-Holland), 1985.

[33] J. A. George and J. W-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[34] J. A. George and J. W-H. Liu. Householder reflections versus Givens rotations in sparse orthogonal decomposition. *Linear Algebra and its Appl.*, 88/89:223–238, 1987.

[35] J. A. George, J. W-H. Liu, and E. G-Y. Ng. A data structure for sparse QR and LU factors. *SIAM J. Sci. Stat. Comput.*, 9:100–121, 1988.

[36] J. A. George, J. W-H. Liu, and E. G-Y. Ng. Row ordering schemes for sparse Givens transformations: I. Bipartite graph model. *Linear Algebra and its Appl.*, 61:55–81, 1984.

[37] J. A. George, J. W-H. Liu, and E. G-Y. Ng. Row ordering schemes for sparse Givens transformations: II. Implicit graph model. *Linear Algebra and its Appl.*, 75:203–224, 1986.

[38] J. A. George, J. W-H. Liu, and E. G-Y. Ng. Row ordering schemes for sparse Givens

transformations: III. Analysis for a model problem. *Linear Algebra and its Appl.*, 75:225–240, 1986.

[39] J. A. George and E. G-Y. Ng. On row and column orderings for sparse least squares problems. *SIAM J. Numer. Anal.*, 20:326–344, 1983.

[40] J. A. George and E. G-Y. Ng. On the complexity of sparse QR and LU factorization of finite element matrices. 1987. (submitted to SIAM J. Sci. Stat. Comput.).

[41] J. A. George and E. G-Y. Ng. Orthogonal reduction of sparse matrices to upper triangular form using Householder transformations. *SIAM J. Sci. Stat. Comput.*, 7:460–472, 1986.

[42] W. Givens. Computation of plane unitary rotations transforming a general matrix to triangular form. *J. Soc. Ind. Appl. Math.*, 6:26–50, 1958.

[43] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins Press, 1983.

[44] G. H. Golub. Numerical methods for solving linear least squares problems. *Numer. Math.*, 7:206–216, 1965.

[45] G. H. Golub and R. J. Plemmons. Large-scale geodetic least squares adjustment by dissection and orthogonal decomposition. *Linear Algebra and its Appl.*, 34:3–27, 1980.

[46] G. H. Golub, R. J. Plemmons, and A. Sameh. *Parallel block schemes for large scale least squares computations.* Technical Report CSRD Rpt. No. 574, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801-2932, April 1986.

[47] S. Hammarling. A note on modifications to the Givens plane rotation. *J. Inst. Maths. Appl.*, 13:215–218, 1974.

[48] M. T. Heath, editor. *Hypercube Multiprocessors 1986*, SIAM, Philadelphia, 1986.

[49] M. T. Heath, editor. *Hypercube Multiprocessors 1987*, SIAM, Philadelphia, 1987.

[50] M. T. Heath. Numerical methods for large sparse linear least squares problems. *SIAM J. Sci. Stat. Comput.*, 26:497–513, 1984.

[51] M. T. Heath. *Parallel Cholesky factorization in message passing multiprocessor environments.* Technical Report ORNL-6150, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, 1985.

[52] M. T. Heath and C. H. Romine. *Parallel solution of triangular systems on distributed memory-multiprocessors.* Technical Report ORNL/TM-10384, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, 1987.

[53] M. T. Heath and D. C. Sorensen. A pipelined Givens method for computing the QR factorization of a sparse matrix. *Linear Algebra and its Appl.*, 77:189–203, 1986.

[54] D. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20:740–777, 1978.

[55] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Adam Hilger Ltd., Bristol, Great Britain, 1981.

[56] A. S. Householder. Unitary triangularization of a nonsymmetric matrix. *J. ACM*, 5:339–342, 1958.

[57] J. A. G. Jess and H. G. M. Kees. A data structure for parallel L/U decomposition. *IEEE Trans. Comput.*, C-31:231–239, 1982.

[58] H. Jordan. A special purpose architecture for finite element analysis. In *Proc. 1978 Int. Conf. Parallel Processing*, pages 263–266, IEEE Computer Soc. Press, 1978.

[59] J. W-H. Liu. An adaptive general sparse out-of-core Cholesky factorization scheme. *SIAM J. Sci. Stat. Comput.*, 7, 1986. (to appear).

[60] J. W-H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Trans. on Math. Software*, 12:127–148, 1986.

[61] J. W-H. Liu. On general row merging schemes for sparse Givens transformations. *SIAM J. Sci. Stat. Comput.*, 7:1190–1211, 1986.

[62] J. W-H. Liu. *On general row merging schemes for sparse Givens transformations.* Technical Report CS-83-04, Dept. of Computer Science, York University, North York, Ontario, July 1983.

[63] J. W-H. Liu. *Reordering sparse matrices for parallel elimination.* Technical Report CS-87-01, Dept. of Computer Science, York University, 1987.

[64] R. E. Lord, J. S. Kowalik, and S. P. Kumar. Solving linear algebraic equations on an MIMD computer. *J. Assoc. Comput. Mach.*, 30:103–117, 1983.

[65] F. T. Luk. A rotation method for computing the QR-decomposition. *SIAM J. Sci. Stat. Comput.*, 7:452–459, 1986.

[66] J. J. Modi and M. R. B. Clarke. An Alternative Givens Ordering. *Numer. Math.*, 43:83–90, 1984.

[67] J. M. Ortega and R. B. Voigt. *A bibliography on parallel and vector numerical algorithms.* Technical Report NASA Contract Report 178335, ICASE, NASA Langley Research Center, Hampton, Virginia, July 1987.

[68] O. Osterby and Z. Zlatev. Direct methods for sparse matrices. In *Lecture Notes in Computer Science 157*, Springer Verlag, Berlin, 1983.

[69] D. Parkynson. Using the ICL DAP. *Comput. Physics Comm.*, 26:227–232, 1982.

[70] F. J. Peters. *Parallel pivoting algorithms for sparse symmetric matrices.* Technical Report, Dept of Math and Computer Science, Eindhoven University of Technology, 1982.

[71] A. Pothen and P. Raghavan. *Distributed orthogonal factorization: Givens and Householder algorithms.* Technical Report, Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, July 1987.

[72] A. Pothen, J. Somesh, and U. Vemulapati. Orthogonal factorization on a distributed memory multiprocessor. In M. T. Heath, editor, *Proc. Hypercube Multiprocessors 1987*, pages 587–596, SIAM, Philadelphia, PA, 1987.

[73] F. Preparata and J. Vuillemin. The cube-connected-cycles: a versatile network for parallel computation. *Comm. Assoc. Comput. Mach.*, 24:300–319, 1981.

[74] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

[75] J. R. Rice. Experiments on Gram-Schmidt orthogonalization. *Math. Comp.*, 20:325–328, 1966.

[76] A. Sameh. On some parallel algorithms on a ring of processors. *Computer Physics Communications*, 37:159–166, 1985.

[77] A. Sameh. An overview of parallel algorithms. *Bull. EDF C1*, 129–134, 1983.

[78] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25:81–91, 1978.

[79] A. H. Sameh and D. J. Kuck. Parallel direct linear system solvers - a survey. In M. Feilmeier, editor, *Parallel Computers - Parallel Mathematics*, International Association for Mathematics and Computers in Simulation, Amsterdam, 1977.

[80] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Trans. on Math Software*, 8:256–276, 1982.

[81] R. Schreiber. *Systolic linear algebra machines: a survery.* Technical Report 87-18, Rensselaer Polytechnic Institute, Troy, New York, June 1987.

[82] L. Snyder. Introduction to the configurable, highly parallel computer. *IEEE Computer*, 15:47–56, Jan. 1982.

[83] D. C. Sorensen. *Analysis of pairwise pivoting in Gaussian elimination.* Technical Report ANL/MCS-TM-26, Argonne National Laboratory, Argonne, IL, February 1984.

[84] G. W. Stewart. The economical storage of plane rotations. *Numer. Math.*, 25:137–138, 1976.

[85] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.

[86] D. W. L. Yen and A. V. Kulkarni. The ESL systolic processor for signal and image processing. In *Proc. 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pages 265–272, November 1981.

[87] Z. Zlatev. Comparison of two pivotal strategies in sparse plane rotations. *Comp. and Maths. with Appls.*, 8:119–135, 1982.

[88] E. Zmijewski. *Sparse Cholesky Factorization on a Multiprocessor.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York 14853-7501, August 1987.