Applied Research Laboratory

2

AD-A229 501

Technical Report

PENNSTATE

1 8 5 5

2

The Pennsylvania State University
APPLIED RESEARCH LABORATORY
P.O. Box 30
State College, PA 16804

SOLVING LEAST SQUARES PROBLEMS ON DISTRIBUTED MEMORY MACHINES

by

Udaya Bhaskar V.S. Vemulapati

Technical Report No. TR 90-015
December 1990

Supported by:
Space and Naval Warfare Systems Command

L.R. Hettche, Director
Applied Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

ENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED

LE AND SUBTITLE

OLVING LEAST SQUARES PROBLEMS ON DISTRIBUTED MEMORY
ACHINES

5. FUNDING NUMBERS

THOR(S)

Jdaya Bhaskar V.S. Vemulapati

FORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Applied Research Laboratory
The Pennsylvania State University
P. O. Box 30
State College, PA 16804

8. PERFORMING ORGANIZATION REPORT NUMBER

INSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Space and Naval Warfare Systems Command
Department of the Navy
Washington, DC 20363-5100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

N000039-88-C-0051

JPPLEMENTARY NOTES

DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited

12b. DISTRIBUTION CODE

BSTRACT *(Maximum 200 words)*

Ve consider solving unconstrained least squares and equality constrained least squares problems on distributed memory multi-processors. First, we examine some issues related to matrix computations in general, on such architecutes. Ve then describe three different algorithms to compute an orthogonal factorizqtion of a matrix on a multi-processor, which are well suited for dense matrices.

As for sparse matrices, efficient solution of problems involving large, sparse natrices on distributed memory multi-processors calls for the use of static lata structures. Often, at the same time, it is critical to detect the rank of a matrix during the factorization to get accurate results. We describe a ange detection strategy, using an incremental condition estimator, that computes a factorization using pre-determined static data structures. We present experimental evidence to show that the accuracy of the rank detection algorithm is comparable to the column pivoting and another recent procedure by Bischof. We further lemonstrate that the algorithm is quite suitable for parallel sparse matrix

(continued on next page)

UBJECT TERMS

Least squares, distributed memory, multi-processors, matrix computations, dense matrices, sparce matrices, hypercule, rank deletion algorithm

15. NUMBER OF PAGES

16. PRICE CODE

ECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

factorizations, by showing good speed-ups on a hypercube with up to 128 processors. We use this algorithm to detect the rank of the constraint matrix in solving the equality constrained least squares problem. We use the weighting approach to solve the equality constrained least squares problem, with two iterations of modified deferred correction technique compute an accurate solution, for all but ill-conditioned problems. We also show that the entire solution process can be carried out using a static data structure. Finally we demonstrate good speed-ups in solving large equality constrained problems on a hypercube.

Abstract

We consider solving unconstrained least squares and equality constrained least squares problems on distributed memory multi-processors. First, we examine some issues related to matrix computations in general, on such architectures. We then describe three different algorithms to compute an orthogonal factorization of a matrix on a multi-processor, which are well suited for dense matrices.

As for sparse matrices, efficient solution of problems involving large, sparse matrices on distributed memory multi-processors calls for the use of static data structures. Often, at the same time, it is critical to detect the rank of a matrix during the factorization to get accurate results. We describe a rank detection strategy, using an incremental condition estimator, that computes a factorization using pre-determined static data structures. We present experimental evidence to show that the accuracy of the rank detection algorithm is comparable to the column pivoting and another recent procedure by Bischof. We further demonstrate that the algorithm is quite suitable for parallel sparse matrix factorizations, by showing good speed-ups on a hypercube with up to 128 processors. We use this algorithm to detect the rank of the constraint matrix in solving the equality constrained least squares problem. We use the weighting approach to solve the equality constrained least squares problem, with two iterations of modified deferred correction technique, to improve the accuracy of the original solution. We present results to indicate that two steps of the modified deferred correction technique compute an accurate solution, for all but ill-conditioned problems. We also show that the entire solution process can be carried out using a static data structure. Finally we demonstrate good speed-ups in solving large equality constrained problems on a hypercube.

(KP)

Table of Contents

List of Tables vii

List of Figures viii

Acknowledgements ix

1 Introduction 1

1.1 Supercomputers 1

1.2 Distributed Memory Multi-processors 2

1.3 Equality Constrained Least Squares 3

1.4 Related Work 4

1.5 Model of Computation 7

1.6 Hypercube Embeddings 8

1.7 Issues in Parallel Matrix Computations 9

1.7.1 Granularity of Parallelism 10

1.7.2 Load Balancing 10

1.7.3 Design Philosophy 11

1.7.4 Experimental Methodology 12

1.7.5 General Techniques 13

1.8 Sparse Matrix Terminology 15

1.9 Overview 16

2 Orthogonal Factorization 18

2.1 Problem Definition 18

2.2 Parallel Givens Rotations 19

2.3 The Knight Tour Givens Sequence 20

2.4 Description of the Algorithm 21

2.5 The Greedy Givens Sequence 24

2.6 The Greedy Algorithm 24

2.7 The Recursive Fine Partition Sequence 26

2.8 Description of the Algorithm 28

2.9 Comparison of the Algorithms 28

3 One-way Dissection 30

3.1 Introduction 30

3.2 Description of Algorithms 31

3.3 Error Analysis 43

3.4 Stability tests 45

3.5 Hypercube Implementation 45

3.5.1 *Complexity of the parallel algorithm* 48

4 Rank Detection 49

4.1 Introduction 49

4.2 Failure of Traditional Methods 50

4.3 Issues in Parallel Factorization 51

4.4 Incremental Condition Estimation 54

4.5 Issues in Parallel Implementation 60

4.6 Implementation Details 62

4.6.1 Compressed and Uncompressed Subscript Notation 62

4.6.2 Static Data Structure for Rank Detection 64

4.7 Timing Results on a Hypercube 65

4.8 Robustness of the Algorithm 65

5 Weighted and Equality Constrained Least Squares Problems 70

5.1 Traditional Methods 70
5.1.1 Using the Basis of a Null Space 71
5.1.2 Direct Elimination 72
5.1.3 Weighting 73
5.2 Deferred Correction 74
5.3 Proper Choice for Parallel Implementation 76
5.4 Issues in Parallel Implementation 76
5.5 Tests on Parallel Implementation 78
5.6 Effectiveness of Deferred Correction 79

Bibliography **81**

List of Tables

3.1 Error in Algorithms 3.3 and 3.5 for Random Matrices 45
3.2 Timings results on Intel Hypercube 47

4.1 Timing results on factorization with the condition estimator 66
4.2 Our condition estimation tests 67
4.3 LINPACK condition estimation tests 67
4.4 Bischof's condition estimation tests 67
4.5 Rank detection tests of our algorithm ($k_2 = 1.0e6$) 68
4.6 Rank detection tests of Bischof's algorithm ($k_2 = 1.0e6$) 69
4.7 Rank detection tests of Column Pivoting algorithm ($k_2 = 1.0e6$) 69
4.8 Rank detection tests of our algorithm ($k_2 = 1.0e9$) 69
4.9 Rank detection tests of Bischof's algorithm ($k_2 = 1.0e9$) 69
4.10 Rank detection tests of Column Pivoting algorithm ($k_2 = 1.0e9$) 69

5.1 Timing results of constrained problem 79
5.2 Trace of the Deferred Correction Algorithm 80

List of Figures

1.1 Connectivity of Hypercubes 8

1.2 Effect of a sequence of Given's rotations on the non-zero structure 16

2.1 The knight tour Givens sequence 20

2.2 The knight tour Givens sequence algorithm 22

2.3 The recursive elimination phase 26

2.4 The fine partition of a matrix 27

4.1 Example matrix R 63

4.2 Uncompressed data storage scheme 63

4.3 Compressed data storage scheme 63

Chapter 1

Introduction

Ever since electronic computers were invented, there has been a constant quest to make them faster and cheaper at the same time. During the past 40 years, there has been tremendous technological developments in the field of integrating the electronic components to build a computer system. Beginning with Small Scale Integration (SSI), the technology has taken us through Medium Scale Integration (MSI), Large Scale Integration (LSI), Very Large Scale Integration (VLSI) and we are going to see Ultra Large Scale Integration (ULSI) in the near future. As a consequence, today we have computer systems that occupy less space by several orders of magnitude, faster by a factor of 10^3–10^4 and yet cheaper by a factor of 1000, compared to a system built in early 1960's.

At the same time, the computers are being called upon to solve bigger and bigger problems each day. It was always felt that there are problems that could use faster computers, however fast the current system might be. Moreover, scientists and engineers are attempting to solve huge problems, which were not attempted before. For example, the need for solving dense linear systems with 20,000 equations, with complex coefficients, arises in radar cross section modeling. In a recent attempt to solve such problems, Scott et al. [69], anticipates the need to solve even larger ones, perhaps with 100,000 equations.

1.1 Supercomputers

In a broad sense, there have been two schools of thought on building such large computer systems capable of solving huge problems. The traditional approach has been to build a single, giant, von Neumann type processor, with a large memory to achieve the goal. Techniques employed in constructing such a machine include ultra fast components, pipelining, multiple arithmetic units, instruction and data caching and vector processing. However, there are two serious drawbacks to this approach. Firstly, physical limitations, such as the speed of light, put an upper bound on how fast the processor can be and in that sense, we can not expect to see the type of increases in the speed of the processors as we have seen in the past 40 years. Secondly, these types of systems are very expensive to build. The second approach has been to build systems with multiple processors and use all the processors to work on the same problem. Experimental machines were built in the early 60's, such as ILLIAC IV. But they never became commercially viable or successful.

In late 70's and early 80's, however, multi-processor systems emerged as a cost-effective competition to the uniprocessor systems. Commercial examples of such systems include BBN Butterfly and Alliant. These systems had relatively small number of processors (typically 4 to 32), that shared a common memory through some hard-wired interconnection. They can be broadly classified into *Shared memory multi-processors.*

1.2 Distributed Memory Multi-processors

In early 1980's, the first prototype of a *Distributed memory multi-processor*, called *Cosmic Cube*, was built at Caltech [70]. This project demonstrated practical feasibility and effectiveness of such machines and it renewed researcher's interests to consider those architectures seriously. The idea behind these systems is very simple. Several (possibly identical) processors, each with its own memory, are connected by suitable communication channels. Each processor can access its (local) memory, without contention from other processors and it can not directly access the memory of other processors. However, the processors can communicate with each other by receiving and sending messages. As the name implies, the memory is distributed between all the processors. They are also referred to as "local memory multi-processors," because the memory is local to each processor. They are also called as *Message-Passing architectures.* Commercial products based on this prototype, known as Hypercubes, have been available in the market since 1985. The notable feature of this architecture are that these systems are easily scalable, as opposed to shared memory machines – i.e. it is not very clear how to build shared memory machines with the order of 1000 processors, where as hypercubes with 8096 processors are already announced. However, not everyone is convinced that these machines are going to be the "Supercomputers of the Future." But the author believes this machines of that type, perhaps with thousands (or even millions) of processors can be built and used for solving large scientific problems.

1.3 Equality Constrained Least Squares

Any problem with sufficient data to over-determine a solution calls for some type of approximation method. And *least squares* is the most frequently used approximation criterion. A well known example is to fit a straight line (curve fitting) through a given set of points in a plane, such that the sum of the squares of the distances between each point and the line is minimized. Mathematically, the problem is to find an n-vector x such that

$$\min_x \|b - Ax\|_2 \tag{1.1}$$

where $b \in \mathbf{R}^m, A \in \mathbf{R}^{m \times n}$ are the given input data. The problem is called the linear least squares problem [59]. In addition, if some of the variables are required to satisfy specified linear constrains, then the problem is called *equality constrained least squares*, usually denoted as to find $x \in \mathbf{R}^n$ such that

$$\min_x \|b - Ax\|_2 \tag{1.2}$$

while

$$Cx = d.$$

These problems occur in many practical engineering problems — optimal design of structures, constrained optimization.

The theory behind the solution of these problems is well understood for small, dense problems for sequential computations [59]. Efficient methods for solving the problem when the matrix $B = \begin{pmatrix} A \\ C \end{pmatrix}$ is large and sparse were given by Van Loan [73], Barlow and Handy [6]. However the problem of solving such large and sparse instances on multiprocessors poses a lot of interesting challenges and this thesis attempts to examine some issues involved in such a process.

To solve those problems efficiently on multi-processors, especially on distributed memory multi-processors (see section 1.2), the proper use of data structures to represent the sparse matrix factorizations is critical. In this context we put some what an extra emphasis on the use of static data structures (see section 1.8) because the use of dynamic data structures (see section 1.8) on distributed memory multi-processors is not very efficient.

1.4 Related Work

As this thesis places an emphasis in demonstrating that such large and sparse problems can be solved efficiently on large distributed memory multi-processors, it borrows quite a few ideas, algorithms and data structures from the sequential counterparts. Here we take a quick look at the the relevant work in the related areas which influenced the author.

Parallel processing in general and distributed memory machines in particular, have attracted a lot of attention of researchers in the last 5 years. Initially there was skepticism among some members of the scientific community about the feasibility of solving real life problems on distributed memory multi-processors. In fact, in November 1985, Alan Karp challenged the scientific community to demonstrate a speed-up of at least 200 for real scientific application on a general purpose MIMD computer [28]. And until that challenge was finally answered by Gustafson et al. [47] in 1988, most of the research in parallel scientific computing was limited to exploring the new architecture and to demonstrating speed-ups on a small number of processors. A good survey of these initial works can be found in McBryan and Van De Velde [62].

Most of the work in the area of parallel matrix computations falls into two main categories. Some earlier work examined the issues involving dense matrices. Chamberlin and Powell [19] described an algorithm to compute a QR decomposition of a matrix on a hypercube. Pothen, Jha and Vemulapati [65] described three algorithms to do the same, one of which is quite similar to the algorithm described in Chamberlin and Powell [19]. Chu and George [21] described an algorithm for computing QR factorization of a rectangular dense matrix on a hypercube, using redundant computations. Geist and Romine [33] investigated the effect of data-storage schemes and pivoting scheme on the efficiency of LU factorization on distributed memory /mp. These algorithms concentrated on the issues of data mapping, processor embedding and to some extent, load balancing and they exploited the architectural aspects of the hypercube.

As far as parallel sparse matrix computations, some ideas from sequential counterparts had a strong influence on them. Some initial work examined the symmetric LU factorization (Cholesky) that set the trend for later development. The important contribution was the concept of symbolic factorization of a sparse matrix to arrive at a static data structure for carrying out the actual factorization. Sherman [71] and Rose et al. [67] gave some practical algorithms to carry out the symbolic factorization for symmetric matrices and later George and Liu [38] gave an optimal algorithm in terms of space and time requirements for the same.

A similar concept of using a static data structure for solving a least squares (using Givens rotations, see Chapter 2) problem was examined by George and Heath [36], making use of the fact that the triangular factor R in a QR factorization of a matrix A is mathematically equivalent to the Cholesky factor of A^TA. Even though this assumption gives us an overestimate of the structure of R, it can be carried out quite economically in practice. Later, however, Coleman et al. [23] characterized a class of matrices for which the estimate is exact and suggested a way wherein an arbitrary matrix can be transformed (using row and column permutations) into block diagonal form, where each block satisfies the characterizing criterion. In practice, one rarely performs such a transformation because the effort involved in doing so does not pay for itself in terms of the computational savings because of lesser number of non-zeros.

Then George and Ng [41] described a way to even store the orthogonal matrix Q in factored form, if Householder transformations are used to compute the factorization. We will be using this result to solve the *equality constrained least squares problems.*

Another problem that attracted attention was to deal with rank deficient matrices. Because we are dealing with sparse matrices, the techniques like column pivoting are not very suitable, especially if static data structures are to be used. Heath [51] described a restricted pivoting procedure, which we call *threshold pivoting*, that allowed the use of static data structures to do some selection based on the actual numerical values. But as we will see later this did not solve all the problems associated with rank deficient matrices. Later George and Ng [40] developed a sparse matrix subroutine package, called SPARSPAK-B, for solving least squares related problems, incorporating most of the known ideas at that time.

In terms of the actual solution of solving least squares problems on multi-processors, George, Heath, Liu and Ng [37] discussed the problem of factoring a large, sparse, positive definite matrix on a multi-processor, with a view to design an algorithm that exploits parallelism, rather than exploiting features of the underlying topology of the hardware.

Other related work has been the solution of triangular system of equations on distributed memory multi-processors. This was not a trivial problem to implement efficiently because it has inherently limited parallelism and has less computational demand. However, by exploiting the architectural aspects of the hypercubes, Heath and Romine [52] discussed several parallel algorithms for solving triangular systems of linear equations on distributed memory multi-processors. Li and Coleman [60] described another column oriented parallel triangular solver and later provided some improvements to the same algorithm [61].

In sharp contrast to the above work, Alaghband [2] described a parallel algorithm for factoring large, sparse, unsymmetric matrices, which dynamically controlled the fill-in with numerical stability as a goal and a dynamic load distribution. This technique is not a pre-ordering of the sparse matrix and is applied dynamically as the factorization proceeds.

1.5 Model of Computation

As has been explained in the previous section, we assume that the system under consideration is a *distributed memory multi-processor*. For conducting experiments, we used Intel Hypercube (both iPSC/1 and iPSC/2) models. The system consists of $P = 2^d$ independent processors, each with its own local memory. d is called the *dimension* of the cube. The interconnection network can be viewed as if a processor (also called as a *node*) sits in each corner of a d-dimensional cube and two processors are connected if and only if there exists an edge between them in the cube. Figure 1.1 shows the interconnection network for $d = 1, 2, 3$ and 4. Inductively, a d-dimensional cube can be constructed by taking two $(d-1)$-dimensional cubes and connecting all the corresponding vertices. It fits the MIMD (Multiple Instructions Multiple Data) model of parallel computations. There is also a separate processor, called the *host*, which is connected to all the nodes in the system by direct communication channels. This *host* acts as a resource manager for the whole system. It allocates some (or all) nodes to a problem on request and loads the programs onto the nodes but it is not involved in node-to-node communications.

We assume that the system supports the following communication primitives. One is to send messages from one node to another. The other is to receive any messages that were sent. Even though every node is not directly connected to every other node, messages can be sent from any node to any other node and the underlying node support system routes the messages appropriately (if needed). In practice, the systems support asynchronous as

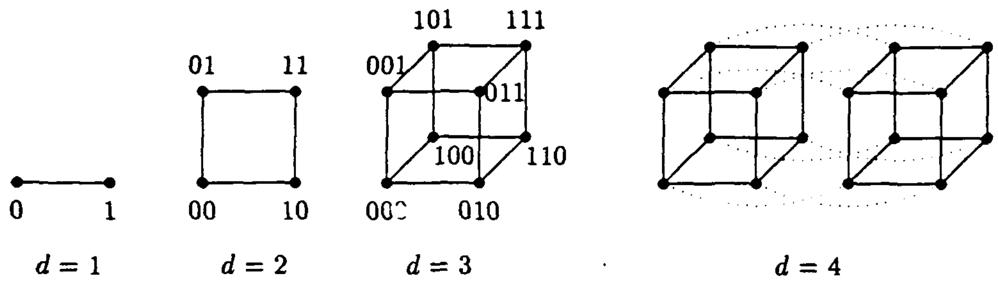


Figure 1.1: Connectivity of Hypercubes

well as synchronous passage of messages. It can be seen easily that the distance between any two nodes is less than equal to $d = \log P$.

1.6 Hypercube Embeddings

As was noted in the previous section, every node is not directly connected to every other node. This means that communication between neighboring nodes (or adjacent nodes) on the cube, is going to be faster than communication between two arbitrary nodes that are not connected. So if the problem is distributed among nodes such that only neighboring nodes need to communicate, the communication delays would be reduced. So a majority of the algorithms designed for Hypercubes try to use only the neighboring communications. However, this is not a serious constraint on the part of the designer. Although arbitrary graphs can not be efficiently embedded on a Hypercube, fairly simple ones like rings, two dimensional grids and trees can be embedded easily. A lot of work has been done on graph embeddings on hypercubes [25, 55, 58], but we limit our discussion to embedding simple graphs like rings and grids.

In the algorithms that are going to be described in this thesis, we make use of only rings and square grids. On a Hypercube with P nodes, two nodes are connected if and only if their node number (numbered from 0 to $(P-1)$) differs in exactly one bit in the binary representation. The *Binary Reflected Gray Codes* are used to number the nodes, we can easily form a ring of P nodes on an a P-node cube. The i^{th} entry, in a *Binary Reflected Gray Code* sequence can be computed by the formula

$$i \oplus (i/2) \quad \text{for } i = 0, 1, 2, \ldots, (P-1),$$

where $\oplus$ is the *exclusive or* operator and the division is integer division. For example, on a cube with 8 nodes, we could order the processors as

000 001 011 010 110 111 101 100

such that the consecutive nodes, including the first and the last are connected on the cube. By a similar extension, square grids of size $\sqrt{P} \times \sqrt{P}$ can be embedded easily on a cube of size P. In practice, each row and column of the grid is a ring.

1.7 Issues in Parallel Matrix Computations

The three basic problems in matrix computations, namely the solution of linear system of equations, linear least squares problems and the eigenvalue problems [43,75], are quite in rich in terms of arithmetic operations and hence ideal candidates for parallelization. At the same time, there are some operational dependencies inherently present in the algorithm, in the sense that some operations are needed to be completed before others can be performed. A major work in designing parallel algorithms is to identify the parts of the solution process that can be done in parallel and maximize such parallelism.

But there are a lot of issues, some still unresolved, when it comes to designing parallel algorithms. Some of the issues that pertain to the algorithms described in this thesis and those that influenced the design are enumerated here.

1.7.1 Granularity of Parallelism

Because the communication cost (the time needed to send one word from one processor to another) on Local-memory multi-processors is quite expensive compared to arithmetic cost (the time required to perform an arithmetic operation on one word), it is imperative that the amount of communication be kept at its minimum. This implies that the ratio of arithmetic to communication cost should be maximized. Philosophically, the communication cost is an necessary evil that arises as a result of trying to exploit parallelism and it should be minimized. Toward this goal, medium to coarse grain parallelism seems to be a good choice and it is reflected clearly in all of the algorithms described here.

1.7.2 Load Balancing

All the algorithms here try to do static load balancing, as opposed to dynamically balancing the load during the execution of the program. Even though a lot of work is being done in this area of dynamic load balancing, it was deliberately decided to use static load balancing to achieve nearly even distribution of work. The main reason is that the overhead involved in dynamic load balancing is high, especially for distributed memory multi-processors.

Even the *static* approach that was taken here is overly simplistic. It is assumed that equal (or nearly equal) distribution of matrix elements (either by rows, columns or sub-matrices) will nearly balance the load on individual processors. It is only fair to note that this is not an unrealistic assumption, if the matrix is carefully distributed among the processors. For example , if the matrix is distributed by rows on a ring of processors, wrapping the rows, instead of blocking them, would achieve a fairly good balance of load. Since wrapping is the the most often used technique in our algorithms, we elaborate on it here.

Suppose we want to distribute a matrix by rows on a ring of processors. Assume that the input matrix has m rows and that there are P processors numbered from 0 to $(P-1)$. In wrapped mapping, row 1 would be assigned to processor 0, row 2 to processor 1 ...etc. And after assigning row P to processor to $(P-1)$, we assign row $(P+1)$ back to processor 0. We continue in this manner till all the columns are distributed. It is easy to note that the difference in number of rows allocated to different processors differs by at most one and that each processor has at least one row from any set of P consecutive rows of the original matrix.

1.7.3 Design Philosophy

There are quite a few metrics that are used to measure the performance of a parallel algorithm running on a parallel processor. Some of them are *speedup, scaled speedup, processor efficiency, price-performance ratio* and a recent one *measured serial fraction,* introduced by Karp and Flatt [56]. Gustafson [46] argues that the *Amdahl's Law* [3] and his argument (about the maximum speedup attainable) are inappropriate for the current approach to massive ensemble parallelism. Further, we agree with Gustafson's quote

> One does not take a fixed-sized problem and run it on various number of processors except when doing academic research; in practice, *the problem size scales with the number of processors*[1].

Even though, in most of the experiments that were done, a known sized problem is run on various number of processors to demonstrate the speedup, as has been noted by Gustafson, it is only of academic interest. A careful examination of the algorithms here will reveal that they are designed for scaled speedup, even though experiments were done only on a fixed size problems. The underlying philosophy has been that these algorithms should do better as the problem size grows, if the number of processors is kept constant. And most algorithms here do not show any increase in the speedup, if the number of

[1]©1988 ACM, Communications of the ACM, May 1988; copied by the permission of the Association for Computing Machinery.

processors is increased beyond a limit. Simply put, it is assumed that the problem size grows with the number of processors. This contrasts sharply with a lot of theoretical work done in parallel processing, where in they assume that $O(n)$ or $O(n^2)$ processors are available, where n is the size of the matrix.

1.7.4 Experimental Methodology

In reporting the experimental results in this thesis, *speedup* is often highlighted as the metric for showing the performance of a parallel algorithm. However, there are a lot of ways to define this metric. Quinn [66] cites the following definitions.

- the ratio between the time taken by a parallel computer executing the fastest serial algorithm and the same parallel computer executing the parallel algorithm using multiple processors.
- the ratio of the execution time of the most efficient serial algorithm running on the fastest serial computer and the execution time of the parallel algorithm running on the parallel computer.
- the ratio of the time taken by a parallel algorithm on a parallel computer using only one processor to the time taken by the same algorithm using multiple processors.

However, raw numbers, in terms of number of seconds taken to solve a particular size problem, do not indicate the effectiveness of the parallel algorithm. Often these numbers may indicate that a 32-node hypercube can not compete with the current minicomputers, in terms of raw times. This is understandable, since these machines were mainly for research purposes and newer versions of these machines not only compete, but they also beat some so called supercomputers. Secondly, it is not fair to compare these raw timing values with the *the best possible* sequential time. Because there has not been enough time and effort spent on these new parallel algorithms (compared to sequential case) to optimize them for a particular architecture. Rather, what is interesting to monitor is how well the algorithm fares as problem size increases, keeping the the same number of processors. And for academic interest, *speedup* was shown as the factor in decrease in time when the number of processors is doubled.

1.7.5 General Techniques

As has been noted before, time required to send one word from one processor to another is significantly greater than the time required for an arithmetic operation. This is true of current generation hypercubes and there is no reason to believe that this gap will be closed any time soon.

However there are a few techniques, which, when employed properly, allow us to solve the problem efficiently. Some notables ones are

- Longer messages

  The time required to send a message of N words from one processor to another can be modeled with the following equation

$$(\alpha + \beta N)\delta$$

  where α is the ***start-up*** time for setting up the message transfer (and is independent of N), β is the time required to send one word from one node to its neighbor after the start-up procedure and δ is the distance between the source and the destination. Even on the recent versions, where dedicated communication handlers exist on each of the nodes to facilitate message routing, the communication delay could be modeled as

$$\alpha_1 + \beta N + (\alpha_2 + \beta N)h$$

  where α_1 is the start-up time at the source/destination and α_2 is the start-up time at each of the intermediate hops and $h = \delta - 1$ is the number of intermediate hops. It is easy to see that the amortized cost of sending one word is reduced considerably if the message is long. Hence short messages should be grouped together and sent as a single, long message.

- Pipelining

  This is another strategy that reduces the total execution time. Even though it takes considerable amount of time to send messages from one node to another, if a series of messages are sent in a pipelined manner, except for the initial delay of $(\alpha_1 + \alpha_2 h)$ units, the messages should be arriving one after the other (at least theoretically). For example, during a QR factorization of a matrix, short messages describing a sequence of Given's rotations could be pipelined and good speed-ups can be achieved, in spite of sending a large number of short messages. This strategy of overlapping communication of messages with computations is possible on the second generation cubes because each node on the hypercube has a dedicated communication handler, which is different from the node processor. There are several ways to achieve this overlap. One can send (and receive) messages asynchronously. While sending messages asynchronously does not save any time in practice, receiving asynchronously would save considerable time if proper choice is exercised. A good strategy with this type of protocol would be to keep checking for a message arrival, followed by some computational work in a repeated fashion.

- Duplication of computations and related asynchrony

  There are situations where in a node makes a binary decision depending on certain values held by all the processors at each step. Traditional approaches used broadcasting of that value to each node to arrive at a consensus value, which requires $O(\log N)$ message delay with N processors. However, by paying a small price in terms of computation, we can postpone resolving that decision for a few steps and instead maintain all possible values for that variable. While doing so, we need not synchronize with other processors at each step and the additional computation is nothing compared to the waiting time involved if we were to wait for the values from all the nodes to arrive.

1.8 Sparse Matrix Terminology

Throughout this thesis, we will be using some well understood terminology from the sparse matrix computations and we take a moment here to review them briefly.

A matrix is *sparse* if the number of nonzero elements in the matrix is quite small compared to the total number of elements. And since the computation time for any dense matrix factorization is of $O(n^3)$ for an $n \times n$ matrix, a lot of savings in time and space can be achieved by not storing the zeros of the sparse matrix.

If the zeros are not be stored, then we need a special structure to represent the sparse matrix. And that data structure needs to keep track of two things — the values of the nonzero elements and the positions of all the nonzeros. The former are the actual numerical values and the later represent the structure of the nonzero pattern. This extra information, which usually requires space of the order of the number of nonzeros, is not required in the case of dense matrices because every element is represented. We could use linked lists or one-dimensional arrays to represent these structures [1]. If we use linked lists, any dynamic changes to the nonzero pattern can be accommodated into the structure.

Consider a matrix, such as shown in figure 1.2, where each x denotes a nonzero. Regardless of the actual values of these nonzeros, one can arrive at a structure that holds all the nonzeros of the final factor, if some factorization is performed on the matrix. This process is called *symbolic factorization* [39, 67]. This process does not take into account the effect of numerical zeros i.e. because of some particular numerical values, some of the entries may become zero during the factorization but this process considers them as structurally nonzero. The figure 1.2 shows the effect of a sequence of Given's rotations

$$\begin{pmatrix} x & 0 & x \\ x & x & 0 \\ x & 0 & x \end{pmatrix} \rightarrow \begin{pmatrix} x & f & x \\ 0 & x & f \\ x & 0 & x \end{pmatrix} \rightarrow \begin{pmatrix} x & f & x \\ 0 & x & f \\ 0 & i & x \end{pmatrix} \rightarrow \begin{pmatrix} x & f & x \\ 0 & x & f \\ 0 & 0 & x \end{pmatrix}$$

Figure 1.2: Effect of a sequence of Given's rotations on the non-zero structure

applied to a given matrix. The f entries also represent nonzeros but were absent in the original matrix and hence called *fill elements.* The i entries indicate that even though they were zeros at the end but were nonzero during the factorization and hence called *intermediate fill elements.*

If we use a symbolic factorization technique to arrive at a structure that holds all the nonzeros during the entire factorization, then we can use pre-determined one-dimensional arrays to represent the sparse factor (see chapter 4). This type of technique is commonly referred to as *using static data structures.*

On the other hand, if we do not want to perform symbolic factorization, or if it is not possible to do so because of pivoting requirements, we will be forced to use linked lists to represent the matrix as new nonzero elements will have to be introduced during the factorization. This technique is called *using dynamic data structures.* In general, because of indirection involved in accessing an element, dynamic data structures are slower than static counterparts.

1.9 Overview

In chapter 2, we consider three algorithms to compute an orthogonal factorization of a rectangular matrix on a hypercube. These techniques are mainly for dense matrices but can be generalized for sparse matrices as well. In chapter 3, we consider solving a sparse system of linear equations on a hypercube. But here the sparse matrix has a special structure, that usually arises out of one-way dissection. In chapter 4, we develop a rank detection technique using an incremental condition estimator that is suitable for parallel sparse matrix factorizations. We examine the need for such an estimator, describe the algorithm, show its effectiveness and demonstrate its usefulness on hypercubes. In chapter 5, we consider solving Equality Constrained Least Squares Problems on hypercubes, using the incremental condition estimator developed in chapter 4 and demonstrate that large and sparse equality constrained least squares problems can be solved efficiently on hypercubes. We conclude with some future directions.

Chapter 2

Orthogonal Factorization

Orthogonal factorization is one of the fundamental operations in matrix computations. Even though using orthogonal factorizations is one of the several ways to solve a system of linear equations, applying them to linear least squares problems is the most practical way to solve them. In this chapter, we design and analyze algorithms for computing orthogonal factorization on a distributed memory multi-processor. A detailed description of these algorithms and analysis can be found in Pothen et al. [65].

2.1 Problem Definition

Given a matrix $A \in \mathbf{R}^{m \times n}$, we would like to obtain the factorization of the form

$$A = QR \tag{2.1}$$

where $Q \in \mathbf{R}^{m \times m}$ is orthogonal and $R \in \mathbf{R}^{m \times n}$ is upper trapezoidal. The form in equation (2.1) is referred to as the QR factorization of the matrix A. It can be calculated in several ways — Gram-Schmidt method, using Householder transformations and using Givens rotations, the last two being more popular [43].

Householder matrices, which are of the form $(I - \frac{2}{u^T u} u u^T)$, are orthogonal for any vector u and they can be used to zero any sub-column of a matrix by a proper choice of u. And the orthogonal factorization can be computed a product of a sequence of Householder transformations.

Givens rotations allow us to zero elements more selectively compared to Householder transformations. Givens rotations are rank-two corrections to the identity of the form

$$J(i,k,\theta) = \begin{array}{cc} & \begin{array}{ccccc} & i & & k & \end{array} \\ \begin{array}{c} \\ i \\ \\ k \\ \\ \end{array} & \begin{pmatrix} 1 & \vdots & & \vdots & 0 \\ \cdots & c & \cdots & s & \cdots \\ & \vdots & & \vdots & \\ \cdots & -s & \cdots & c & \cdots \\ & \vdots & & \vdots & \end{pmatrix} \end{array} \tag{2.2}$$

where $c = cos(\theta)$ and $s = sin(\theta)$ for some θ. Clearly Givens rotations are orthogonal for any value of θ. Premultiplication by $J(i,k,\theta)$ amounts to a rotation of θ degrees in the (i,k) coordinate plane. In fact if $x \in \mathbf{R}^n$ and $y = J(i,k,\theta)x$, then

$$\begin{aligned} y_i &= cx_i + sx_i \\ y_k &= -sx_i + cx_k \\ y_j &= x_j \qquad\qquad j \neq i \text{ or } k. \end{aligned} \tag{2.3}$$

Hence by choosing

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}} \text{ and } s = \frac{x_k}{\sqrt{x_i^2 + x_k^2}}, \tag{2.4}$$

we can set y_k to zero. And the orthogonal matrix Q in equation (2.1) can be computed by a product of a sequence of Givens rotations.

2.2 Parallel Givens Rotations

We only consider Givens sequences in which zeros once created are preserved during the rest of the factorization. Givens rotations are very promising in the parallel context, since disjoint rotations can potentially be computed in parallel. There is also a great deal of freedom in the order in which the rotations are applied to the matrix. This freedom can be exploited to design algorithms for appropriate architectures. Here we study three

$$\begin{array}{cccc}
X & X & X & X \\
\boxed{6} & X & X & X \\
\boxed{5} & \boxed{7} & X & X \\
\boxed{4} & \boxed{6} & \boxed{8} & X \\
\boxed{3} & \boxed{5} & \boxed{7} & \boxed{9} \\
\boxed{2} & \boxed{4} & \boxed{6} & \boxed{8} \\
\boxed{1} & \boxed{3} & \boxed{5} & \boxed{7}
\end{array}$$

Figure 2.1: The knight tour Givens sequence

different Givens sequences.

- The *Knight tour* Givens sequence, discussed by Gentleman [34], who showed that the grouping of rotations in this sequence led to tighter error bounds, and by Sameh and Kuck [68], who designed a parallel orthogonalization algorithm for a SIMD machines using this sequence.
- The Greedy Givens sequence, in which rotations are organized into groups such that as many disjoint rotations as possible are put into each group. It was studied by Modi and Clarke [63] and by Cosnard, Robert and Muller [24]. This is a variant of an algorithm designed and implemented by Chamberlain and Powell [19]
- The recursive fine partition sequence, in which the matrix is partitioned into submatrices and these submatrices are distributed among the processors.

2.3 The Knight Tour Givens Sequence

The name comes from the similarity between this sequence and the way a knight moves on a chess board. In this algorithm, an element a_{ij} is eliminated by a rotation between rows i and $(i-1)$.We denote this operation by a tuple (i,j). The sequence is illustrated in figure 2.1 for a 7×4 matrix. The squares indicate that the entry is to be zeroed and the integer inside the square indicates the group number in which that element is zeroed. All elements in a group can be zeroed concurrently. For an $m \times n$ matrix, there are $(m+n-2)$ groups in our sequence.

The entire sequence can be divided into two phases — an *increasing phase* when the number of rotations in each successive group increases (more precisely, does not decrease) and a *decreasing phase* when the number of rotations decreases in each successive phase. In the example shown in figure 2.1, groups 1 through 7 belong to the increasing phase and groups 8,9 and 10 to the decreasing phase.

To implement this algorithm, we assume that the processors numbered 0 to $(P-1)$ form a ring. We can then define a *predecessor* of a processor k as $(k-1) \bmod P$ and *successor* as $(k+1) \bmod P$. The rows of the matrix are numbered 1 to m and they are mapped onto processors by wrapping; row m is stored on processor 0, row $(m-1)$ on processor 1,...etc. Each processor then gets approximately $\frac{n}{P}$ rows. We also define a *top row* of a processor as the lowest numbered row it holds and the *bottom row* as its highest numbered row.

2.4 Description of the Algorithm

Zeros are introduced in the matrix from the bottom to top, and from left to right. To zero an element in row i, a processor receives row $(i-1)$ from its *successor*, computes the rotation and updates row i. Concurrently, its successor receives row i, computes the rotation and updates row $(i-1)$. Since we used a ring formation of the nodes on the hypercube, the processor which communicate are always neighbors. The node program is illustrated in figure 2.2.

The variable *col* corresponds to the column position in row i in which a zero is introduced by the Givens rotation. The algorithm uses the concept of *active* rows. If the rotation $(i,1)$ belongs to group k, then the row i becomes active only when the processor

```
repeat as long as active rows exist
    for each active row
        /* zero $(i, col)$ */
        send row $i$ to predecessor
        recv row $i - 1$ from predecessor
        update row $i$

        /* help zero $(i + 1, col)$ */
        send row $i$ to successor
        recv row $i + 1$ from successor
        update row $i$
    endfor
end_repeat
```

Figure 2.2: The knight tour Givens sequence algorithm

it is on has completed all rotations in groups smaller than k. In other words, a row becomes *active* when its column element can be zeroed by a rotation in the group currently executed by the processor. Once a row becomes active, it remains active, until the row is completely processed.

Initially, the only active row on a processor is its *bottom* row. By the way the knight tour Givens Sequence is defined, rotations $(i, j+1)$ and $(i-2j, 1)$ belong to the same group. Hence the algorithm can deterministically decide when to make its next row active. i.e. if a node holds a row i, its next higher numbered row will be $(i - P)$ and hence that row becomes active after $\lceil \frac{P}{2} \rceil$ zeros have been introduced in row i.

During each iteration, the node program makes new rows active and updates the count of number of active rows. In the increasing phase, this number increases. Once the *top row* has been processed, the program enters the decreasing phase and it terminates when the number of active rows becomes zero. Only the flow of data has been shown in figure 2.2 and all the above details are not shown.

We also note that there is no need for explicit synchronous messages between processors, even though at a given instant different processors may be executing rotations belonging to different groups. This is possible because each processors keeps track of the next column in which a zero can be introduced. It also keeps track of the column up to which its *predecessor* has zeroed using this row. Thus each processor can infer the state of its predecessor as well as its successor from the rows sent and received.

This sequence has been used a great deal for systolic arrays, which are based entirely on nearest neighbor communication [7,35,53].

To analyze the complexity of this algorithm, we bound the time required for a group of rotations in the sequence. This estimate can only be an upper bound on the complexity of the algorithm, since in the parallel algorithm that was described above, computations in different groups can overlap.

The maximum number of rotations in a group is n, one in each column. Hence the number of rows in a group that need to be updated by a processor is no more than $X = \lceil \frac{2n}{p} \rceil$. When a zero is introduced in column j, there are $(n-j)$ elements in that row that need to be updated. Let $\pi = \frac{p}{2}$. Since the rows are wrapped onto the processors, the number of elements that a processor updates in a group is bounded by

$$\nu = n + (n-\pi) + (n-2\pi) + \ldots + (n-(X-1)\pi) = nX - \frac{\pi}{2}X(X-1).$$

On simplifying, we get $\nu = \frac{n^2}{p} + \frac{n}{2}$. The number ν is also an upper bound on the number of elements a processor needs to communicate to its neighbor for the rotations in a group.

Updating a row segment of j elements requires $2j$ flops on a processor. Each group hence takes time less than 2ν flops. Since there are $(m+n-2)$ groups in the whole matrix, the arithmetic complexity is bounded by $2\nu(m+n-2)$, which is approximately $\frac{2n^2}{p}(m+n-2)$.

By a similar reasoning, the communication time can be shown to be equal to $\frac{2n^2}{p}(m+$

$n-2)\alpha + \frac{2n}{P}(m+n-2)\beta$, where α and β are as defined in section (1.7.5).

For a square matrix of order n, the arithmetic time reduces to $\frac{5n^3}{3P}$. Note that the coefficient of the leading term should be optimally $\frac{4}{3}$ and hence this algorithm is not *optimal*. The degradation of the performance can be attributed to the duplication of the work involved in same rotation being computed on both processors.

2.5 The Greedy Givens Sequence

In this algorithm, the matrix is again distributed by rows among the processors. And in each partial column that a processor holds, it zeros all but one of the entries that need to be zeroed, using only the rows that it holds. That is why it is called *greedy*. This can be done by all the processors simultaneously. After that the processors cooperate in a *recursive elimination phase* to zero the remaining elements in that column. And since there is communication only in the second phase, the communication overhead in this algorithm is low.

As in the previous algorithm, a ring of P processors is assumed. However, during the second phase, we also make use of the other interconnections of the hypercube to make sure that the nodes that communicate with each other are neighbors on the hypercube. A minor variant of this algorithm has been described and implemented in Chamberlain and Powell [19], although it was fine-tuned using several architectural features to optimize the running time.

The nodes are numbered from 0 to $(P-1)$ and the rows of the matrix are numbered from 0 to $(m-1)$. The first n rows are wrapped among the P processors but the rest of the rows are equally distributed among the processors in any manner.

2.6 The Greedy Algorithm

Each column of the the original matrix is transformed into a column of the triangular matrix in two phases — an internal rotation phase and a recursive elimination phase. The algorithm essentially computes one column at a time.

Let us consider the transformation of column j of the matrix. It is assumed that the columns numbered 0 to $(j-1)$ have already been completely transformed. As a consequence, the rows 0 to $(j-1)$ need not be updated any more. Without loss of generality, let us assume that the row j is being held by processor 0. The *top row* on a processor at this stage is the lowest numbered row numbered greater than or equal to j.

In the internal rotation phase, each processor zeros elements in column j that need to be eliminated, using the rows that it holds. At the end of this phase, there will only be one nonzero element in the top row in column j on each node.

During the second phase, called *recursive elimination phase*, all the processors cooperate to eliminate the rest of the elements in column j. This proceeds in $\log P$ steps. In each step two processors exchange their top rows and carry out an elimination. At the end of $\log P$ steps, all the necessary elements in column j would have been zeroed.

In k-th step of this phase, processors that differ in their numbers in the k-th most significant bit pair up to perform the elimination. The processor that appears later in the Gray code ring zeros its element and the other processor updates its row. Because of the way the ring of processors is formed using Binary Reflected Gray Codes (see section 1.6, page 9), all processors that pair up during this entire phase are neighbors. The pairs of processors that pair up in each stage is illustrated in figure 2.3, for a case of $P = 8$. This diagram makes an assumption that processor 0 holds the diagonal element. In practice, it is not difficult for the nodes to figure out the current holder of the diagonal row and pair up accordingly.

Chamberlain and Powell's algorithm differs from the above in only one sense. During the k-th step, processors that differ in their node numbers in their k-th least significant bit pair up to do the recursive elimination.

To arrive at the complexity of this algorithm, consider the processing of column j. During the internal rotation phase, each processor zeros $\lceil (m-j)/P \rceil - 1$ elements and

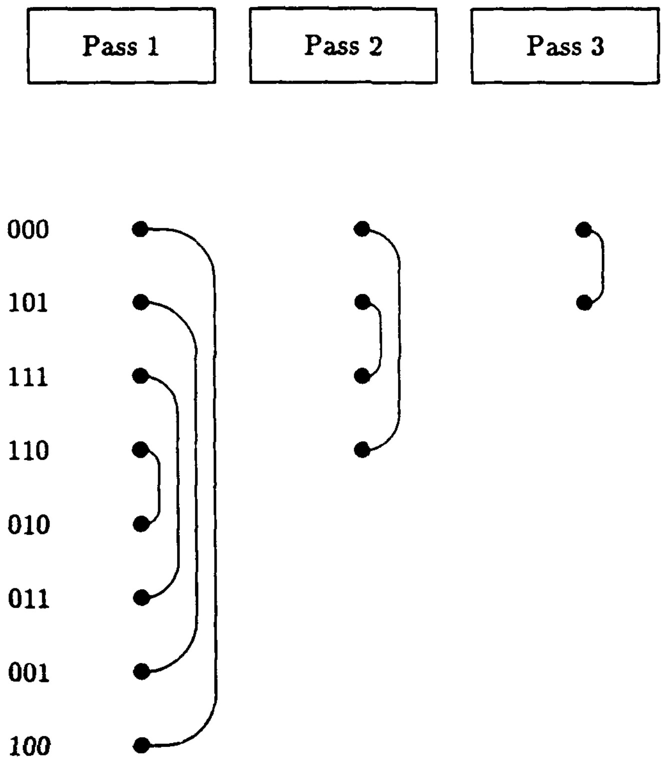


Figure 2.3: The recursive elimination phase

each rotation involves $4(n-j)$ flops, since two rows of length $(n-j)$ are involved. Summing the cost over all the columns, we get the arithmetic cost of this phase to be $\frac{2n^2}{P}(m-n/3)$.

In the recursive elimination phase, each processors updates at most one row in each step. Since there are $\log P$ steps, we get the cost of this phase to be $2(n-j)\log P$ for column j. Summed over all columns, it comes to $n^2 \log P$. Hence the arithmetic complexity of this algorithm is $\frac{2n^2}{P}(m-n/3)+n^2\log P$. Similarly the communication complexity can be shown to be $\frac{n^2}{2}(\log P)\alpha + n(\log P)\beta$.

2.7 The Recursive Fine Partition Sequence

For ease of exposition, we consider a square matrix of order n. We also assume that the P processors form a square grid of size $p \times p$ (see section 1.6). The processors form a

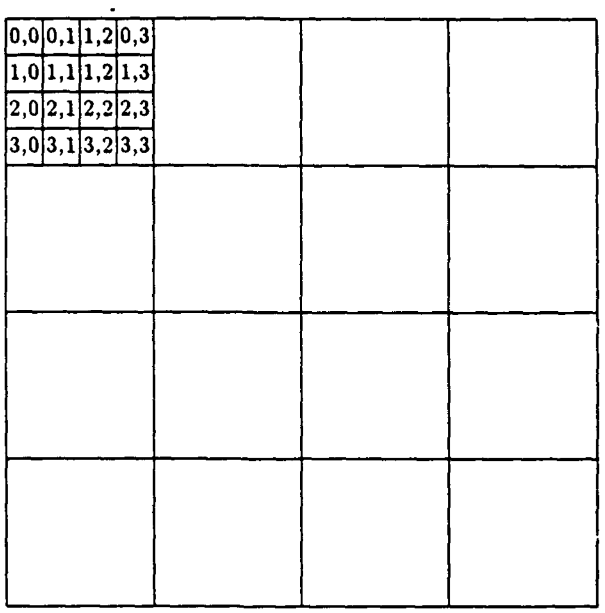


Figure 2.4: The fine partition of a matrix

ring in each row and in each *column of the grid*. It is interesting to note that each row or column of processors is a subcube of dimension $\log p$. We also denote by p_{ij}, a processor on the i-th row and j-th column of the grid, where $0 \leq i, j \leq (p-1)$. We call this the processor grid.

At the same time, the matrix is coarsely divided into p *column blocks* and p *row blocks*, to give us p^2 coarse submatrices, each square of order $\frac{n}{p}$. These column and row blocks are numbered from 0 to $(p-1)$. Each coarse submatrix is further divided into p *column subblocks* and p *row subblocks*, which partition it in to p^2 *fine submatrices*. Thus each submatrix is square of order $\eta = n/p^2$. Within each coarse submatrix, denote by (i, j) the fine fine submatrix formed by the i-th row subblock and j-th column subblock, where $0 \leq i, j \leq (p-1)$. Now the fine submatrix (i, j) is assigned to the processor p_{ij}. The partitioning and the mapping are illustrated in figure 2.4.

2.8 Description of the Algorithm

The general idea behind this algorithm also involves two steps. In the first step, called the *diagonalization phase*, each processor diagonalizes a column subblock it holds, using the submatrix that it holds. It also sends these transformations to the other processors in that row so that they can apply these transformations to their subblocks. In the second phase, called *recursive elimination phase*, is very similar to the one described in the previous algorithm, except that we deal with subblocks of a matrix rather than a single column. More details of this algorithm can be found in Pothen et al. [65].

The arithmetic complexity can be shown to be

$$\frac{4n^3}{3P} + \frac{n^3 \log P}{2P^{1.5}}.$$

And the communication complexity is

$$\frac{n^2}{P}\alpha + \frac{n^2 \log^2 P}{4P}\alpha + \frac{n \log^2 P}{4}\beta.$$

2.9 Comparison of the Algorithms

As has been noted before, the knight tour givens sequence is not optimal in terms of arithmetic complexity. But the last two algorithms are asymptotically optimal in that sense, as the leading term for a square matrix is $\frac{4n^3}{3P}$.

Regarding the communication complexity, there have been some results on the lower bound on the communication requirements for parallel Cholesky factorization [57]. However we know of no such results for parallel orthogonal factorization. It is conjectured that the lower bound for that would be $\frac{n^2}{P} \log P$, the $\log P$ factor coming because of the difference in Cholesky and orthogonal factorizations. In this sense the greedy algorithms seems to be optimal. Even the recursive fine partitioning algorithm, with the communication complexity of $O(\frac{n^2}{P} \log^2 P)$, is only marginally greater than the trivial lower bound

$\frac{n^2}{P}$.

Chapter 3

One-way Dissection

In this chapter, we consider solving a sparse system of linear equations, where the matrix has a special structure that arises out of the so-called one-way dissection ordering. We suggest a solution process for this problem that makes it trivially parallelizable, especially on distributed memory multi-processors. The error analysis results as well as many of the algorithmic details are due to Barlow and are cited here for the sake of completeness. A detailed account of this problem may be found in Barlow and Vemulapati [9].

3.1 Introduction

The basic problem is to solve the $n \times n$ system of linear equations

$$Ax = s, \tag{3.1}$$

where A and s have the form

$$A = \begin{pmatrix} B_1 & 0 & 0 & \cdot & S_1 \\ 0 & B_2 & 0 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 0 & \cdot \\ \cdot & \cdot & \cdot & B_k & S_k \\ G_1^T & G_2^T & \cdot & G_k^T & F \end{pmatrix}; \quad s = (s_1, s_2, \ldots, s_k, s_{k+1})^T. \tag{3.2}$$

Here $B_i, i = 1, 2, \ldots, k$, are $m_i \times m_i$ matrices, F is a $p \times p$ matrix and G_i and S_i are $m_i \times p$ matrices, where $p + \sum_{i=1}^{k} m_i = n$. Each $s_i, i = 1, 2, \ldots, k$, is an m_i-vector and s_{k+1} is a p-vector. This is a matrix that arises out of the so-called one-way dissection ordering. Much of the discussion of one-way dissection in the literature has concerned symmetric, positive definite systems. This implies, that $B_i, i = 1, 2, \ldots, k$, and F are symmetric, positive definite, and $G_i = S_i, i = 1, 2, \ldots, k$. Instead we make the much weaker assumption that $rank(A) = n$, i.e. , that A is nonsingular. Thus we have that $rank(B_i) = l_i \leq m_i, i = 1, 2, \ldots, k$. Applications of such systems are given in Gunzberger and Nicholaides [45].

Gunzberger and Nicholaides [44] suggested an algorithm based upon Gaussian elimination with singular pivots. It uses the Moore-Penrose inverses of the diagonal blocks $B_i, i = 1, 2, \ldots, k$. The Moore-Penrose pseudoinverse of a matrix B, denoted by B^+, is the unique matrix satisfying the four Moore-Penrose conditions

$$\begin{array}{ll} 1.\ BB^+B = B & 3.\ (BB^+)^T = BB^+ \\ 2.\ B^+BB^+ = B^+ & 4.\ (B^+B)^T = B^+B \end{array} \tag{3.3}$$

We will use the notation $B^{(i)}$,$B^{(i,j)}$, or $B^{(i,j,k)}$ to denote matrices satisfying conditions i,j, or k among those in (3.3). Their algorithm [44] has a simple elimination procedure, but a complicated back substitution procedure.

Here we suggest an alternative method for resolving the singularity in the diagonal blocks $B_i, i = 1, 2, \ldots, k$. This method is based upon the weighted pseudoinverse discussed in a fundamental paper by Elden [29]. We give evidence that this method is more stable. We also give a more elegant back substitution procedure, which makes the algorithm easier to implement on a message passing architecture. These algorithms are outlined in section 3.2. Empirical tests verifying the stability properties of our algorithm are given in section 3.4. We also give an implementation on Intel Hypercube(iPSC/1) in section 3.5.

3.2 Description of Algorithms

We first describe the elimination procedure of Gunzberger and Nicholaides [44] for solving (3.1). It makes use of the Moore-Penrose pseudoinverses of the diagonal blocks

$B_i, i = 1, 2, \ldots, k$. The other elimination procedures in this section will take a similar form.

Algorithm 3.1 *Block Elimination using the Moore-Penrose Pseudoinverse [44]*

1. *Compute*

$$\tilde{F} = F - \sum_{i=1}^{k} G_i^T B_i^+ S_i \ ; \quad \tilde{s}_{k+1} = s_{k+1} - \sum_{i=1}^{k} G_i^T B_i^+ s_i$$

$\tilde{G}_i^T = G_i^T(I - B_i^+ B_i)$ *projection of* G_i^T *onto orthogonal complement of Range*(B_i)

2. *For* $i = 1, 2, \ldots, k$ *find an* $m_i \times (m_i - l_i)$ *matrix* X_i *such that* $B_i X_i = 0$. *Note that* $l_i = rank(B_i)$. *Thus* X_i *is a basis for the null space of* B_i. *Algorithms for finding such a basis are given by Heath [51] and Pothen [64].*

We note that the terms $G_i^T B_i^+ S_i$, $G_i^T B_i^+ s_i$, $i = 1, 2, \ldots, k$, can be computed independently, as can the null space bases $X_i, i = 1, 2, \ldots, k$. The same is true for the $\tilde{G}_i^T, i = 1, 2, \ldots, k$, but we will see later that it is not necessary to compute these matrices at all.

The back substitution phase of the Gunzberger-Nicholaides procedure is somewhat complicated. Let $x = (x_1, x_2, \ldots, x_k, x_{k+1})^T$, where the components $x_i, i = 1, 2, \ldots, k, k+1$, are of the form

$$x_i = y_i + z_i, \quad \text{where } y_i^T z_i = 0, i = 1, 2, \ldots, k, k+1, \tag{3.1}$$

and the vectors z_i satisfy

$$B_i z_i = 0, \quad i = 1, 2, \ldots, k \quad , \tag{3.2a}$$

$$\tilde{F} z_{k+1} = 0 \quad . \tag{3.2b}$$

Since A is nonsingular, $\tilde{F}$ is also nonsingular (cf. [44]). Thus

$$z_{k+1} = 0 \quad , \tag{3.3a}$$

$$x_{k+1} = y_{k+1} \quad . \tag{3.3b}$$

Then Algorithm 1 reduces (3.1) to the system

$$B_i y_i + S_i x_{k+1} = s_i, \quad i = 1, 2, \ldots, k, \tag{3.4a}$$

$$\sum_{i=1}^{k} \tilde{G}_i^T z_i + \tilde{F} x_{k+1} = \tilde{s}_{k+1}. \tag{3.4b}$$

From (3.2a), $\tilde{G}_i^T z_i = G_i^T z_i, i = 1, 2, \ldots, k$, thus we can replace (3.4b) with

$$\sum_{i=1}^{k} G_i^T z_i + \tilde{F} x_{k+1} = \tilde{s}_{k+1}. \tag{3.5}$$

Thus $\tilde{G}_i^T$ need never be explicitly computed. The system (3.4) can be written

$$My = \tilde{s} - Nz \tag{3.6}$$

where

$$M = \begin{pmatrix} B_1 & 0 & \cdot & 0 & S_1 \\ 0 & B_2 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & B_k & S_k \\ 0 & \cdot & \cdot & 0 & \tilde{F} \end{pmatrix} \tag{3.7a}$$

$$N = \begin{pmatrix} 0 & \cdot & \cdot & \cdot & S_1 \\ 0 & \cdot & \cdot & 0 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & S_k \\ G_1^T & G_2^T & \cdot & G_k^T & 0 \end{pmatrix} \tag{3.7b}$$

$$\tilde{s} = (s_1, s_2, \ldots, s_k, \tilde{s}_{k+1})^T;$$

$$y = (y_1, \ldots, y_k, y_{k+1})^T;$$

$$z = (z_1, \ldots, z_k, z_{k+1})^T \tag{3.7c}$$

The consistency of (3.6) and the nonsingularity of $\tilde{F}$ follow from the nonsingularity of A. If we assume that z is known, and let

$$f = (f_1, f_2, \ldots, f_k, f_{k+1})^T = \tilde{s} - Nz \tag{3.8}$$

then a basic (non-unique) solution y is given by

$$y_{k+1} = x_{k+1} = \tilde{F}^{-1} f_{k+1} \tag{3.9a}$$

$$y_i = B_i^+ (f_i - S_i y_{k+1}) \tag{3.9b}$$

From [44], we have that y solves (3.6). Thus if we define the matrix Φ such that

$$y = \Phi f, \tag{3.10}$$

where Φ has the form

$$\Phi = \begin{pmatrix} B_1^+ & 0 & \cdot & \cdot & -B_1^+ S_1 \tilde{F}^{-1} \\ \cdot & B_2^+ & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & B_k^+ & -B_k^+ S_k \tilde{F}^{-1} \\ 0 & \cdot & \cdot & 0 & \tilde{F}^{-1} \end{pmatrix}, \tag{3.11}$$

by combining (3.6) and (3.10) we have

$$(I - M\Phi)Nz = (I - M\Phi)\bar{s}. \tag{3.12}$$

Let

$$X = diag(X_1, X_2, \ldots, X_k, 0),$$

where X_i are defined in Algorithm 3.1. Thus (3.12) becomes

$$Tw = g \tag{3.13}$$

where $T = (I - M\Phi)NX$; $z = Xw$;$g = (I - M\Phi)\bar{s}$. Equation (3.13) is consistent but overdetermined (cf. [44]). It can be solved by an orthogonal factorization of T (in Gunzberger and Nicholaides [44], the use of normal equations is advocated). Gunzberger and Nicholaides show that T must have full rank if A has full rank. We assume that the dimensions of the null spaces of $B_i, i = 1, 2, \ldots, k$, are much smaller than the dimensions of the blocks themselves. That is, $m_i - l_i \ll m_i$. Thus, the solution of (3.13) should be very fast compared with the rest of the algorithm. We state the procedure as Algorithm 3.2.

Algorithm 3.2 *Back Substitution Procedure [44]*

1. *Explicitly form* $T = (I - M\Phi)NX; g = (I - M\Phi)\bar{s}$.

2. *Solve $Tw = g$ by orthogonal factorization (or normal equations).*

3. *Let $z = Xw$ and solve*

$$y = \Phi(\bar{s} - Nz)$$

4. *The solution $x = y + z$.*

We propose two changes in Algorithms 3.1 and 3.2. The first is a simplification of the back substitution procedure. This simplification uses computations arising directly out of the elimination procedure. To describe that, we give a more specific version of Algorithm 3.1 which includes the method for computing $B_i^+, i = 1, 2, \ldots, k$. The method is slightly different from that given in Gunzberger and Nicholaides [44], but uses the method for computing B_i^+ given in Golub and Van Loan [43].

Algorithm 3.3 *Implementation of Block Elimination using the Moore-Penrose pseudoinverse*

1. *For $i = 1, 2, \ldots, k$ perform steps 2-7.*

2. *Factor B_i into*

$$B_i = Q_i \begin{pmatrix} U_i^{[1]} & U_i^{[2]} \\ 0 & 0 \end{pmatrix} P_i^T$$

where Q_i is orthogonal, $U_i^{[1]}$ is an $l_i \times l_i$ upper triangular, $U_i^{[2]}$ is an $l_i \times (m_i - l_i)$ matrix, and P_i is a permutation matrix. This factorization and the determination of rank l_i can be done by orthogonal decomposition with column pivoting (cf. [59, Chapter 10]) or some other method (cf. [12, 20, 31]).

3. *Compute*

$$\begin{pmatrix} S_i^{[1]} & s_i^{[1]} \\ S_i^{[2]} & s_i^{[2]} \end{pmatrix} = Q_i^T(S_i, s_i)$$

where $S_i^{[1]}$ is $l_i \times p$ and $S_i^{[2]}$ is $(m_i - l_i) \times p$.

4. Solve for $\hat{S}_i$ and $\hat{s}_i$

$$U_i^{[1]}(\hat{S}_i, \hat{s}_i) = (S_i^{[1]}, s^{[1]})$$

5. Compute

$$X_i = \begin{pmatrix} -[U_i^{[1]}]^{-1} U_i^{[2]} \\ I_{m_i - l_i} \end{pmatrix} \tag{3.14}$$

X_i is a common choice for the null basis matrix of B_i (cf.[51, 64]).

6. Factor

$$X_i = Z_i \begin{pmatrix} W_i \\ 0 \end{pmatrix}$$

where Z_i is orthogonal and W_i is upper triangular and compute

$$(V_i, v_i) = Z_i \begin{pmatrix} 0 & 0 \\ 0 & I_{p-m_i+l_i} \end{pmatrix} Z_i^T(\hat{S}_i, \hat{s}_i)$$

The items (V_i, v_i) are projections of $(\hat{S}_i, \hat{s}_i)$ onto the space orthogonal to the null space of B_i, thus providing $B_i^+(S_i, s_i)$.

7. Compute

$$(R_i, r_i) = -G_i^T(V_i, v_i)$$

8. Compute

$$\tilde{F} = F + \sum_{i=1}^{k} R_i; \; \tilde{s}_{k+1} = s_{k+1} + \sum_{i=1}^{k} r_i$$

Algorithm 3.3 requires

$$2m_il_i(m_i - l_i) + \tfrac{2}{3}l_i^3 + 2m_il_i(p+1) + l_i^2(m_i - l_i) +$$
$$(m_i - l_i)^2(m_i - \tfrac{1}{3}(m_i - l_i)) + 4(m_i - l_i)m_i(p+1) + p(p+1)l_i + O(m^2)$$

flops for each $i = 1, 2, \ldots, k$. Let $m = \max_{1 \leq i \leq k} m_i$. If $p \leq m$ and $|m_i - l_i| \leq c = O(1)$ this simplifies to

$$2m_i l_i(p+1) + \frac{2}{3} l_i^3 + p(p+1) l_i + O(cm^2)$$

for each $i = 1, 2, \ldots, k$. We assume here that all of blocks in (3.2) are dense.

If we consider equation (3.4a) and apply the reduction from Algorithm 3.3, we have

$$\begin{aligned} U_i y_i + S_i^{[1]} x_{k+1} &= s_i^{[1]} && (3.15a) \\ S_i^{[2]} x_{k+1} &= s_i^{[2]}. && (3.15b) \end{aligned}$$

where $U_i = (U_i^{[1]}, U_i^{[2]})$. Since equation (3.15) is just an orthogonal reduction of some rows from $Ax = s$, it follows that it is underdetermined but consistent. Using the null basis (3.14) for B_i and by letting

$$\hat{G}_i = G_i^T X_i$$

equation (3.5) becomes

$$\sum_{i=1}^{k} \hat{G}_i w_i + \tilde{F} x_{k+1} = \tilde{s}_{k+1} \qquad (3.16)$$

where $z_i = X_i w_i$. Thus if we let $S^{[2]} = (S_i^{[2]}, \ldots, S_k^{[2]})^T$, $s^{[2]} = (s_i^{[2]}, \ldots, s_k^{[2]})^T$, and $\hat{G} = (\hat{G}_1, \ldots, \hat{G}_k)$, then x_{k+1} and $w = (w_1, w_2, \ldots, w_k)^T$ solve the linear system

$$\begin{pmatrix} \hat{G} & \tilde{F} \\ 0 & S^{[2]} \end{pmatrix} \begin{pmatrix} w \\ x_{k+1} \end{pmatrix} = \begin{pmatrix} \tilde{s}_{k+1} \\ s^{[2]} \end{pmatrix}. \qquad (3.17)$$

The nonsingularity of A guarantees that (3.15) is a nonsingular system of linear equations. For problems arising in practice, its dimension will be small compared to the dimension of A. It can be solved by Gaussian elimination with partial pivoting or orthogonal decomposition. Such a reduction is much simpler than the back substitution procedure in Algorithm 3.2. The values of y_i and x_i, $i = 1, 2, \ldots, k$ can be recovered from (3.9a) and the step

$$x_i = y_i + X_i w_i. \tag{3.18}$$

The computation (3.9a) can be simplified into

$$y_i = [U]_i^+(s_i^{[1]} - S_i^{[1]} x_{k+1}) \tag{3.19}$$

thereby avoiding the reuse of the orthogonal factor Q_i. We now formally state this procedure as Algorithm 3.4. This algorithm is a method for solving (3.5) and is simply a particular implementation of Algorithm 3.2.

Algorithm 3.4 *Improved Back Substitution Procedure*

1. *Solve the linear system in equation (3.17) for x_{k+1} and $w = (w_1, w_2, \ldots, w_k)^T$ using orthogonal factorization by Householder transformations.*

2. *For $i = 1, 2, \ldots, k$, do steps 3–6*

3. *Compute*

$$f_i^{[1]} = s_i^{[1]} - S_i^{[1]} x_{k+1}. \tag{3.20}$$

4. *Let*

$$g_i = -\begin{pmatrix} I_{l_i} \\ 0 \end{pmatrix} [U_i^{[1]}]^{-1} f_i^{[1]}, \tag{3.21}$$

$$\hat{g}_i = (I_{m_i - l_i}, 0) Z_i^T g_i.$$

5. *Solve*

$$\begin{pmatrix} U_i^{[1]} & U_i^{[2]} \\ 0 & W_i \end{pmatrix} y_i = \begin{pmatrix} f_i^{[1]} \\ \hat{g}_i \end{pmatrix}.$$

Here $S_i^{[1]}$, $s_i^{[1]}$, $G_i^{[1]}$, $U_i^{[1]}$, W_i, and Z_i are from Algorithm 3.3.

6. *Compute*

$$x_i = y_i + X_i w_i,$$

where X_i is in Algorithm 3.3.

The back substitution procedure requires

$$\frac{2}{3}[p + \sum_{i=1}^{k}(m_i - l_i)]^3 + \sum_{i=1}^{k}[3l_i(m_i - l_i) + \frac{1}{2}m_i^2 + \frac{1}{2}l_i^2] + O(m)$$

flops. If $\max_{1 \le i \le k} |m_i - l_i| = c = O(1)$ then this reduces to

$$\frac{2}{3}[p + kc]^3 + \frac{1}{2}\sum_{i=1}^{k}[l_i^2 + m_i^2] + O(cm)$$

flops.

The second modification to Algorithms 3.1 and 3.2 is to replace B_i^+ with $B_i^{(1,3)}, i = 1, 2, \ldots, k$, i.e. any matrix $B_i^{(1,3)}$ satisfying Penrose conditions 1 and 3. For the elimination algorithm, this is equivalent to solving (cf. [29])

$$\min_{(V_i, v_i)} \|B_i(V_i, v_i) - (S_i, s_i)\|_F$$

and then computing

$$\tilde{F} = F - \sum_{i=1}^{k} G_i^T V_i, \qquad (3.22a)$$

$$\tilde{s}_{k+1} = s_{k+1} - \sum_{i=1}^{k} G_i^T v_i, \qquad (3.22b)$$

$$\tilde{G}_i^T = G_i^T(I - B_i^{(1,3)} B_i). \qquad (3.22c)$$

It is essential that all of the columns of

$$(H_i, h_i) = (S_i, s_i) - B_i(V_i, v_i) \qquad (3.23)$$

be vectors in the space orthogonal to the columns of B_i. It is guaranteed by the use of $B_i^{(1,3)}$. This allows us to set up equation (3.17) by orthogonal factorization of B_i by column pivoting or some other method to detect rank [12, 20, 31]. When we substitute $B_i^{(1,3)}$ for B_i^+, we lose the property that $y_i^T z_i = 0$, but this property is not necessary for the algorithm to work. Again since $\tilde{G}_i^T z_i = G_i^T z_i$, it is not necessary to do the computation (3.22c).

The matrix $B_i^{(1,3)}$ is not unique unless B_i has full rank. In our modified algorithm, we can choose $B_i^{(1,3)}$ so as to minimize $||G_i^T B_i^{(1,3)} S_i||_F$ and $||G_i^T B_i^{(1,3)} s_i||_2$. As is shown in Barlow and Vemulapati [9], this leads to a new algorithm with better numerical stability properties. Elden [29] showed that the (1,3) pseudoinverse with this property is the weighted pseudoinverse defined below.

Definition 3.1 *The G-weighted pseudoinverse of B is defined by*

$$B_G^+ = (I - (G^T P)^+ G^T) B^+$$

where

$$P = I - B^+ B.$$

In Elden [29], it is shown that the matrix B_G^+ is the (1,3)-inverse such that

$$||G^T B_G^+ S||_F \le ||G^T B^{(1,3)} S||_F \tag{3.24}$$

for all (1,3)-inverses of B and matrices S. The G-weighted pseudoinverses $[B_G]_i^+$ need not and should not be explicitly computed. Instead we compute the quantities

$$\begin{aligned} R_i &= -G_i^T [B_G]_i^+ S_i \quad i = 1, 2, \ldots, k, && (3.25a)\\ r_i &= -G_i^T [B_G]_i^+ s_i \quad i = 1, 2, \ldots, k, && (3.25b) \end{aligned}$$

and then compute

$$\tilde{F} = F + \sum_{i=1}^{k} R_i; \quad \tilde{s}_{k+1} = s_{k+1} + \sum_{i=1}^{k} r_i. \tag{3.26}$$

The quantities (R_i, r_i) are simply the residuals of the least squares problem

$$\min_{(V_i,v_i)\in T_{B_i}} ||G_i^T(V_i, v_i)||_F; \tag{3.27}$$

where T_{B_i} is the set of minimizers of

$$\min_{(V_i,v_i)\in \mathbf{R}^{m_i\times(p+1)}} ||B_i(V_i, v_i) - (S_i, s_i)||_F.$$

The computation of (V_i, v_i) is not necessary. The residuals (R_i, r_i) can be computed directly. The problem (3.27) has an unique solution if *rank* $\begin{pmatrix} B_i \\ G_i^T \end{pmatrix} = m_i, i = 1, 2, \ldots, k$. This is a direct consequence of nonsingularity of A. We now give a more detailed description of this procedure. Steps 1-4 are the Björck-Golub [18] direct elimination procedure for solving (3.27).

Algorithm 3.5 *Block Elimination Scheme Using the weighted Pseudoinverse*

1. *For* $i = 1, 2, \ldots, k$, *do steps 2-5*

2. *Same as Steps 2-3 of Algorithm 3.3.*

3. *Let* $G_i^T = (G_i^{[1]}, G_i^{[2]})$ *where* $G_i^{[1]}$ *is a* $p \times l_i$ *matrix and* $G_i^{[2]}$ *is a* $p \times (m_i - l_i)$ *matrix. Compute* $\hat{G}_i = G_i^{[2]} - G_i^{[1]}[U_i^{[1]}]^{-1}U_i^{[2]}$ *and* $(\hat{S}_i, \hat{s}_i) = -G_i^{[1]}[U_i^{[1]}]^{-1}(S_i, s_i)$. *(Note that* $\hat{G}_i = G_i^T X_i$*).*

4. *Factor*

$$\hat{G}_i = Z_i \begin{pmatrix} W_i \\ 0 \end{pmatrix},$$

where Z_i is orthogonal and W_i is upper triangular. Then compute

$$(R_i, r_i) = Z_i \begin{pmatrix} 0 & 0 \\ 0 & I_{p-n_i+l_i} \end{pmatrix} Z_i^T(\hat{S}_i, \hat{s}_i).$$

5. *Compute*

$$\bar{F} = F + \sum_{i=1}^{k} R_i; \quad \bar{s}_{k+1} = s_{k+1} + \sum_{i=1}^{k} r_i.$$

With the change that

$$g_i = -G_i^{[1]}[U_i^{[1]}]^{-1} f_i^{[1]} \tag{3.28}$$

in (3.21), the back substitution procedure in Algorithm 3.4 can be used directly after Algorithm 3.5. This adds an additional $l_i p^*$flops for each $i = 1, 2, \ldots, k$. Except for differences in terms of $O(m^2)$, the operation count for Algorithm 3.5 is identical to that of Algorithm 3.3. We note however, one difference that the matrix

$$\bar{\Phi} = \begin{pmatrix} B_1^{(1,3)} & 0 & \cdot & \cdot & -B_1^{(1,3)} S_1 \bar{F}^{-1} \\ \cdot & B_2^{(1,3)} & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & B_k^{(1,3)} & -B_k^{(1,3)} S_k \bar{F}^{-1} \\ 0 & \cdot & \cdot & 0 & \bar{F}^{-1} \end{pmatrix}$$

is only a (1)-pseudoinverse of M. This can be verified easily. However this is enough to assure that $y = \bar{\Phi} f$ satisfies (3.10) . Hence we can use the back substitution procedure in Algorithm 3.4.

3.3 Error Analysis

The following error analysis results, due to Barlow [9], are cited below for the sake of completeness.

Theorem 3.1 *Let Algorithm 3 or 5 be implemented using Householder transformations in floating point arithmetic with machine unit μ. Let the backward substitution phase be done using Algorithm 3.4. Then the computed solution $\bar{x}$ satisfies*

$$(A + \delta A)\bar{x} = s + \delta s$$

where

$$||\delta A||_F \leq \phi_A \; \tau_A ||A||_F \, \mu + O(\mu^2)$$

$$||\delta s||_2 \leq \phi_s \; \tau_A ||s||_2 \, \mu + O(\mu^2)$$

$$\tau_A = \max\{ \max_{1 \leq i \leq k} ||G_i^{[1]} [U_i^{[1]}]^{-1}||_2, \max_{1 \leq i \leq k} ||G_i B_i^{(1,3)}||_2 \}$$

and ϕ_A and ϕ_s are modestly sized polynomials in the dimension of A.

We now give a corollary that gives stronger stability results for Algorithm 3.5. It is a straightforward consequence of Theorem 3.1

Corollary 3.1 *Let Algorithm 5 be implemented using Householder transformations in floating point arithmetic with machine unit μ. Then δA and δs in Theorem 3.1 satisfy*

$$||\delta A||_F \leq \phi_A \; \tau_G ||A||_F \, \mu + O(\mu^2),$$

$$||\delta s||_2 \leq \phi_s \; \tau_G ||s||_2 \, \mu + O(\mu^2),$$

where

$$\tau_G = \max_{1 \leq i \leq k} ||G_i^{[1]} [U_i^{[1]}]^{-1}||_2,$$

and ϕ_A and ϕ_S are modestly sized polynomials in the dimension of A.

The error bounds obtained by this analysis are better for Algorithm 5 than for Algorithm 3.3. In the next section, we give numerical tests which seem to indicate that

Algorithm 5 gives more reliable answers.

3.4 Stability tests

We implemented Algorithms 3 and 5 in FORTRAN single precision on the SUN3 with the back substitution procedure in Algorithm 3.4. The two algorithms differ only in their computation of $B_i^{(1,3)}, i = 1, 2, \ldots, k$.

The matrix A is generated randomly. Rank one singularities are introduced into each diagonal block by replacing the last row of each such block by the sum of its other rows. Then the right hand side is formed by making the known solution vector $(1, 1, \ldots, 1)^T$. We then calculated the relative error in the solution. The results are shown in Table 3.1. Here the experiments clearly suggest that Algorithm 3.5 has better numerical stability properties than Algorithm 3. Thus we see that the use of the weighted pseudo-inverse rather than the Moore-Penrose pseudoinverse gives us a better method of resolving the singularity in the diagonal blocks.

3.5 Hypercube Implementation

To simplify the implementation on a Hypercube, it is assumed that each diagonal block B_i and F are of equal size i.e. $m_i = p, i = 1, 2, \ldots, k$, and that $p = k + 1$, i.e. the size of each diagonal block is also equal to the number of diagonal blocks. It then follows that

n	k+1	Estimated Condition No.	Error:Alg.3	Error:Alg.5
2	2	2.0E02	0	0
4	2	1.0E01	9.0E-6	6.0E-7
10	2	4.0E01	3.0E-6	2.0E-6
10	3	1.0E02	4.0E-6	2.0E-6
20	4	3.0E02	9.0E-6	3.0E-6
40	5	8.0E02	2.0E-4	4.0E-5
60	6	9.0E02	8.0E-5	7.0E-6
80	8	2.0E03	1.0E-4	2.0E-5
100	10	2.0E04	2.0E-3	5.0E-5

Table 3.1: Error in Algorithms 3.3 and 3.5 for Random Matrices

$p^2 = n$. The number of processors in the Hypercube is denoted by P (numbered from 1 to P). It is further assumed that the number of diagonal blocks $k+1$ is at least as large as the number of processors (P).

The blocks $B_i, i = 1, 2, \ldots, k$ are equally distributed among the first $P-1$ processors, along with the corresponding S_i and G_i matrices. And the matrix F is processed by the node P. A brief description of the algorithm emphasizing the flow of data between the processors follows.

Host Program

```
generate matrix $A$ and the vector $s$
compute the number of blocks that each node numbered from 1 to $P-1$ gets
for $i := 1$ to $P-1$
    send appropriate blocks of $B, S, G$ and $s$ to node $i$
send $F$ to node $P$
wait for the solution parts to arrive from all the nodes
```

Node Program

```
if it is not the last node ($P$) then
    receive the matrix blocks $B, G, S$ and $s$
    diagonalize each $B_i$ and solve the LSE problem as described in Algorithm 3
    send the matrices $\hat{G}_i$ and $S_i^{[2]}$ along with $s_i^{[2]}, R_i$ and $r_i$ to node $P$ (cf. Algorithm 3.5)
    wait for $x_{k+1}$ and $w_i$ vectors to arrive from node $P$
    complete the solution process to get $x_i$
    send $x_i$'s to the host
else
    receive F from the host
```

size of each block(p)	no. of processors(P)	time in seconds
8	8	1.22
8	4	1.46
8	2	2.64
16	16	5.36
16	8	7.86
16	4	11.46
32	32	34.12
32	16	48.62
32	8	71.94

Table 3.2: Timings results on Intel Hypercube

```
receive the matrices $\hat{G}_i$ and $S_i^{[2]}, s_i^{[2]}, R_i$ and $r_i$ sent by all other nodes
solve the system (3.17)
broadcast $x_{k+1}$ and appropriate blocks of $w_i$ to all the other $P-1$ nodes
send $x_{k+1}$ to host
```

The above Algorithm was implemented in FORTRAN on an Intel hypercube (iPSC/1) at the ACRF facility at Argonne National Laboratory and the Table 3.2 shows the timings results from these experiments. The matrix in each case was an $p^2 \times p^2$ matrix. For a fixed value of p, the problem was run on cubes of different dimensions to determine the speed-up. *The time shown is elapsed time in seconds from the moment the host starts sending the data to the nodes till the final solution is returned to the host.*

It appears from the results that by increasing the number of processors by a factor of 2, one would get a speed-up by a factor of 1.43. The main reason is that the back substitution process has a bottleneck — the other nodes must remain idle while node P determines x_{k+1} and w.

3.5.1 Complexity of the parallel algorithm

It is assumed that the time required to transmit a message of N words from one node to another is $(\alpha + \beta N)d$ where α is the start-up time for the message and β is the time required to send one word after the initial message is set-up and d is the distance between the nodes.

The only communication required in the parallel algorithm described above is the transmission of $\hat{G}_i$, $S_i^{[2]}$, $s_i^{[2]}$, R_i and r_i to node P and vectors x, w from node P to nodes 1 to $P-1$. Since the size of R_i is much larger than other matrices and since the maximum distance between any two nodes on the Hypercube is $\log P$, it is easily seen that the upper bound on the communication complexity of the algorithm is $O([P\alpha + \beta(p^2 + Pn)] \log P)$.

The computational complexity is easier to bound because all the computational work except the solution of (3.17) is done in parallel and hence it is divided equally among $P-1$ processors. However the matrix in the system (3.17) is of the order $p + \sum_{i=1}^{n}(m_i - l_i)$ and hence only $\frac{2}{3}(p + \sum_{i=1}^{n}(m_i - l_i))^3$ are not done in parallel.

Chapter 4

Rank Detection

As was noted in the introductory chapter, the problem of dealing with rank deficient matrices during the factorization, especially when using static data structures was not fully explored in the literature. As we will see in the next chapter, accurate rank detection of a constraint matrix becomes very critical to the solution process when we deal with equality constrained least squares problems. In this chapter, we explore ways of detection of rank using an incremental condition estimation technique, which facilitates us to easily solve the equality constrained least squares problem. Iain S. Duff [27] pointed out to Jesse Barlow that this particular rank detection technique can be used effectively on frontal solvers for large, sparse systems of linear equations. Some of preliminary results presented here appeared in Barlow and Vemulapati [10].

4.1 Introduction

Choosing a set of linearly independent columns from a given matrix, within a tolerance of machine precision, is a common subproblem, among problems involving matrix computations. Subtle variations of the same problem are "rank detection" and "condition estimation."

Traditional methods of rank detection for dense matrices include QR factorization with column pivoting [43], the singular value decomposition [43] and a host of condition estimators [54], and the LINPACK 1-norm estimator. The scheme due to Hager and Higham [49] also gained recent acceptance. Threshold pivoting [51] strategy is often used in the case of sparse matrices.

However, when we consider solving these problems on a parallel architectures, most of the traditional approaches fail to be cost effective, especially when large and sparse matrices are involved.

Here we propose an incremental condition estimator, which is quite reliable and is well suited for parallel sparse matrix QR factorizations. In section 4.2, we examine the reasons for the failure of traditional methods when applied to our problem. In section 4.3 we discuss the issues in the effective implementation of solving our problem on a parallel architecture. In section 4.4, we describe the an algorithm that allows us to incrementally estimate the condition number of the triangular factor during the factorization. In section 4.5, we discuss the implementation issues on a parallel architecture and provide experimental results. In section 4.8, we provide experimental evidence that suggests that the algorithm is robust enough.

4.2 Failure of Traditional Methods

The general strategy [36] for doing a QR factorization of a sparse matrix C is

1. Determine the symbolic structure of C^TC.

2. Using a heuristic approach, find a permutation matrix P, such that P^TC^TCP has a sparse cholesky factor.

3. Generate the storage structure for R by doing a symbolic factorization of P^TC^TCP.

4. Compute R numerically.

Although it is known that finding a permutation in step 2, that produces an optimally sparse Cholesky factor is a hard problem (in fact, NP-hard), many good heuristic approaches such as minimum degree and nested dissection give us fill-reducing orderings [39]. This approach of determining the data structures required for the R factor before the actual factorization (in other words a static data structure) has some advantages, compared to dynamically set up storage structures during the factorization. The accessing of the elements in a static set up is faster and hence the factorization step is likely to be faster. Since the static structure does not depend on the numerical values of the original matrix C, the cost involved in steps 1—3 can be spread over a number of factorizations if repeated computations of R are required with different numerical values of C.

Most of the known algorithms for rank detection (or condition estimation) are neither cost effective nor appropriate for sparse matrix applications. Any estimator requiring $O(n^2)$ units of computation time is too expensive for sparse matrices, considering that the factorization of a sparse matrix itself requires only $O(n^{1.5})$. The QR factorization with column pivoting upsets the sparsity pattern, because the column ordering chosen in step 2 is not used. Moreover the pivoting process requires us to use a dynamic data structure for R. The singular value decomposition is too expensive for practical use, even though it is the most accurate algorithm for rank detection.

4.3 Issues in Parallel Factorization

Here we limit our discussion to distributed memory machines, such as Hypercubes, while talking about parallel architectures. If we want to implement the factorization in parallel, we need to re-examine the validity of the traditional methods on such machines. The column pivoting algorithm requires that the processors have to synchronize to select the next pivot column. This introduces not only delays due to communication overheads but also forces the program into a lock-step mode, leaving no room for pipelining ahd / or overlapping of computations. As was observed already, any pivoting process results in more fill-in and hence more computation time.

Dynamic data structures are not easy to distribute in a local-memory environment ; even if we manage to do that, keeping track of the current state of the structure among all processors is not an easy task. Hence we consider using static data structures. The *threshold strategy* described by Heath [51] and implemented in SPARSPAK-B [40] allows us to deal with the static data structures for most of the computations. The following is a brief description of that algorithm.

Algorithm 4.1 *Threshold Pivoting*

```
/* $\epsilon$ is a tolerance factor */
done←false
k←1
l←0
While not done do
    $\gamma \leftarrow \|(c_{l+1,k}, \ldots, c_{s,k})^T\|_2$
    if $\gamma \geq \epsilon$ then
      Construct an orthogonal transformation $H_1 = diag(I_l, \bar{H}_l)$ such that
      $\bar{H}_l(c_{l+1,k}, \ldots, c_{s,k}) = \pm\gamma e_1^{(s-l)}$
      Compute $C \leftarrow H_1 C$
      $p_l \leftarrow k$
      $l \leftarrow l + 1$
    endif
    $k \leftarrow k + 1$
    done $\leftarrow (l \geq s)$ or $(k \geq n)$
endwhile
```

The above algorithm does not create any non-zeros that are not predicted by the George-Heath strategy. Even though empirical tests show that this strategy rarely fails in practice, dramatic failures in rank detection are possible in some cases. A simple example is the following bidiagonal matrix, which will be considered full rank matrix for any value of a, even though D could be arbitrarily ill-conditioned. (In fact, actual $\kappa(D) \approx a^3$).

$$D = \begin{pmatrix} 1 & a & 0 & \dots & 0 \\ 0 & 1 & a & \dots & 0 \\ 0 & 0 & \ddots & a & 0 \\ 0 & \dots & & 1 & a \\ 0 & \dots & & 0 & 1 \end{pmatrix}$$

If we use static data structures, during the factorization, we are only allowed to look at each column only once in a given sequence and we should be able to determine whether a new column is linearly independent of the others already chosen to be in the factor. This translates to checking whether the resulting upper triangular factor is going to be well conditioned.

To this end, Bischof [12] describes an incremental estimator for the smallest singular value, which is a modified 2-norm condition estimator suggested by Cline, Conn and Van Loan [22, 74]. However this algorithm has two serious drawbacks when it comes to sparse matrices. Firstly the estimator requires n^2 flops during the triangularization of an $n \times n$ matrix. Secondly, its estimate of the smallest singular value differs arbitrarily from the actual value, for matrices with special structure. In particular, if the new row being added is orthogonal to the current approximate singular vector, then the estimate is likely to be very poor. As an example, consider the following 3×3 matrix.

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \beta & -\beta\omega & 2 \end{pmatrix}.$$

For a specific value of $\omega = 1 - \sqrt{2}$, the estimate of the smallest singular value from Bischof's algorithm is 1, independent of β, while the actual 2–norm of $D \approx \beta^2$. Such trivial 3×3 examples can be constructed easily for any algorithm that does not use look-ahead in the estimation algorithm. However, the look-ahead aspect does not seem to have any serious effect in most practical situations.

Recently Bischof, Lewis and Pierce [13] extended the original algorithm to handle the case of general matrices and showed how this modified approach can be used for nested dissection case.

4.4 Incremental Condition Estimation

The proposed algorithm is an "incremental ∞–norm" estimator that uses look-ahead. It is a modification of LINPACK 1–norm estimator for upper triangular matrices [26]. The algorithm looks at each column just once and decides whether to include that column in the factorization or not. It does that by incrementally estimating the ∞–norm of the inverse of the partially formed upper triangular factor. There is no column pivoting and hence static data structures can be used.

We are interested in computing a QR factorization of the matrix C, with accurate rank detection, so that the factored matrix has the following form

$$C = Q \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}.$$

Define the sequence of upper triangular matrices $U_1^{(k)}, k = 1, 2, \ldots, l$, by

$$U_1^{(1)} = (u_{11}) \quad \text{where } u_{11} = \|c_1\|_2$$

and

$$U_1^{(k+1)} = \begin{pmatrix} U_1^{(k)} & v_{k+1} \\ 0 & \gamma_{k+1} \end{pmatrix}$$

where

$$v_{k+1} = (c_{1,k+1}^{(k)}, \ldots, c_{k,k+1}^{(k)})^T,$$

$$c_{k+1}^{(k)} = (c_{1,k+1}^{(k)}, \ldots, c_{m_1,k+1}^{(k)})^T$$

is the $(k+1)^{st}$ column of C after $H_1, H_2, \ldots, H_k$ are applied and

$$\begin{aligned}\gamma_{k+1} &= \|(c_{k+1}^{(k)}, \ldots, c_{m_1,k+1}^{(k)})^T\|_2 \\ &= \sqrt{\|c_{k+1}^{(k)}\|_2^2 - \|v_{k+1}\|_2^2}\,.\end{aligned}$$

Let

$$L^{(k)} = [U_1^{(k)}]^T$$

and

$$a^{(1)} = 1;$$

$$\hat{\sigma}_1 = 1/u_{11} = 1/\|c_1\|_2 = 1/\gamma_1.$$

To choose the $(k+1)^{st}$ column, we let $x^{(k)}$ be such that

$$L^{(k)} x^{(k)} = a^{(k)}$$

where $a^{(k)}$ is a vector of ± 1, chosen to maximize $\|x^{(k+1)}\|_\infty$. Then compute

$$x^{(k+1)} = (x^{(k)}, \xi_{k+1})^T$$

where

$$\xi_{k+1} = \gamma_{k+1}^{-1}(-sgn(v_{k+1}^T x^{(k)}) - v_{k+1}^T x^{(k)}).$$

Thus

$$\hat{\sigma}_{k+1} = \max\{\hat{\sigma}_k, |\xi_{k+1}|\} = \|x^{(k+1)}\|_\infty.$$

This procedure is precisely the LINPACK estimator without the "look-ahead" property.

To incorporate a "look-ahead," we consider the partial sums

$$\rho_j^+ = v_j^T \begin{pmatrix} x^{(k)} \\ \xi_{k+1}^+ \end{pmatrix} \text{ and } \rho_j^- = v_j^T \begin{pmatrix} x^{(k)} \\ \xi_{k+1}^- \end{pmatrix}$$

where

$$\xi_{k+1}^+ = \gamma_{k+1}^{-1}(1 - v_{k+1}^T x^{(k)})$$

and

$$\xi_{k+1}^- = \gamma_{k+1}^{-1}(-1 - v_{k+1}^T x^{(k)})$$

and

$$v_j = (c_{1,j}^{(k)}, \ldots, c_{k-1,j}^{(k)})^T$$

is the j^{th} column of C after $H_1, H_2, \ldots, H_k$ are applied. Note that the last entry of v_j is not known until after we use column k to form H_k. The ρ_j can be accumulated through out the computation. For weights $t_1, t_2, \ldots, t_n > 0$, we then examine

$$\zeta^+ = |\xi_{k+1}^+| + \sum_{j \in Nonz(c^{[k]})} t_j \rho_j^+$$

and

$$\zeta^- = |\xi_{k+1}^-| + \sum_{j \in Nonz(c^{[k]})} t_j \rho_j^-$$

and choose

$$x^{(k+1)} = (x^{(k)}, \xi_{k+1}^+)^T$$

or

$$x^{(k+1)} = (x^{(k)}, \xi_{k+1}^-)^T$$

according to whether ζ^+ or ζ^- is larger. The choice of weights is heuristic. LINPACK chooses $t_j = u_{j,j}^{-1}$. However, we have not computed $u_{j,j}$ at this point. So we choose

$t_j = \gamma_j^{-1}$.

We now explain how this can be used in column selection. Our algorithm performs the condition estimator on the most recently formed diagonal block C_{ii} until

1. it finds a zero diagonal
2. the estimate of $\|C_{ii}^{-1}\|$ exceeds ε^{-1} where ε is a predefined tolerance and is usually $O(\mu)$, μ being the machine precision.

In both cases, we restart the condition estimator with all $\rho_i = 0$ (implicitly) and then begin forming $C_{i+1,i+1}$. Of course, in case 2, we must find a dependent column in C_{ii}. To do this, we solve

$$C_{ii}h = e_k.$$

Let ν be the index such that

$$|h_\nu| = \max_{1 \le j \le k} |h_j|.$$

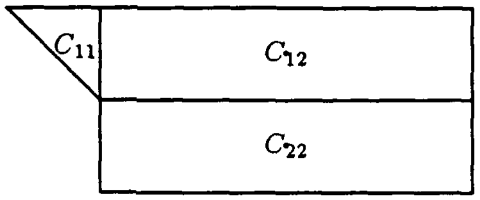
And we delete the column ν from C_{ii} and re-triangularize the new matrix by a sequence of Given's rotations. We can then conclude that the rank of C_{ii} is $(k-1)$. The conclusion is based on the following heuristic. The condition estimator considered the previous set of $(k-1)$ columns to be independent. And the addition of the k-th column does not increase the rank and it also can not decrease it. The reordering of the columns is done to insure that the last entry in the last column is of negligible magnitude. The general idea of the algorithm is detailed below.

Algorithm 4.2 *Rank detection by incremental condition estimation*

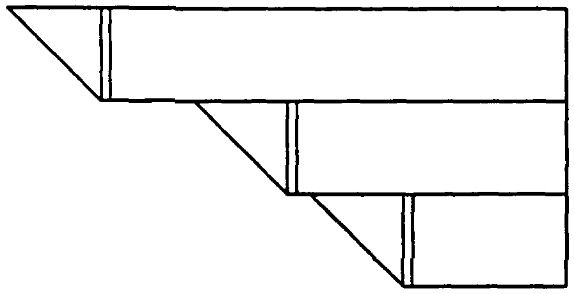
```
/* $\epsilon \approx \mu$ is a tolerance factor. $c^{[l]}$ denotes the $l^{th}$ row of $C$. */
$l \leftarrow 0$
$done \leftarrow false$
$k \leftarrow 1$
```

```
$firstk \leftarrow 1$
$firstl \leftarrow 1$
$q_i \leftarrow i,\ i = 1, 2, \ldots n$
$\rho_i \leftarrow 0,\ i = 1, 2, \ldots n$
$\gamma_i \leftarrow \|c_i\|^2,\ i = 1, 2, \ldots n$
$\sigma \leftarrow 0$
while not done do
$\xi \leftarrow \gamma_k^{-1}(-sign(\rho_k) - \rho_k)$
$\hat{\sigma} \leftarrow max\{\sigma, |\xi|\}$
if $\hat{\sigma}^{-1} > \epsilon$ then
    /* this column is good */
    construct an orthogonal transformation such that the current column is zeroed
    $l \leftarrow l + 1$ and $q_l \leftarrow k$
    for $j \in Nonz(c^{[l]})$
        update the column norms $\gamma_j$
    $\xi^+ \leftarrow \gamma_k^{-1}(1 - \rho_k)$
    $\xi_- \leftarrow \gamma_k^{-1}(-1 - \rho_k)$
    $\zeta_{max}^+ \leftarrow |\xi^+|$
    $\zeta_{max}^- \leftarrow |\xi^-|$
    for $j \in Nonz(c^{[l]})$ update the partial sums
        $\rho_j^+ \leftarrow \rho_j + c_{l+1}\xi^+$
        $\rho_j^- \leftarrow \rho_j + c_{l+1}\xi^-$
        $\zeta_{max}^+ \leftarrow max\{\zeta_{max}^+, |\rho_j^+/\gamma_j|\}$
        $\zeta_{max}^- \leftarrow max\{\zeta_{max}^-, |\rho_j^-/\gamma_j|\}$
    /* We just looked ahead of the affected columns and computed both possible
    values for the partial products */
    if $\zeta^+_{max} \geq \zeta^-_{max}$ then
      $\sigma \leftarrow max\{\sigma, |\xi^+|\}$
      $\rho_j \leftarrow \rho^+_j, \quad j \in Nonz(c^{[l]})$
    else
      $\sigma \leftarrow max\{\sigma, |\xi^-|\}$
      $\rho_j \leftarrow \rho^-_j, \quad j \in Nonz(c^{[l]})$
  else    the column is not good
    if $\gamma_{max} \leq \epsilon$ then
      /* no more good columns and we are done */
      exit
    else
      Let $C_{\omega\omega}$ be the submatrix of $C$ with rows from $firstl$ through $l$
      and columns from $firstk$ through $k$.
      Solve $C_{\omega\omega}h = (0, 0, \ldots, 1)^T$.
      Let $|h_\nu| = \max_i(|h_i|)$
      /* Break ties arbitrarily in choosing the maximum element */
      Move the column $\nu$ to the last column of $C_{\omega\omega}$ and re-triangularize $C_{\omega\omega}$.
      $firstl \leftarrow 1$
      $firstk \leftarrow 1$
      $\rho_i \leftarrow 0$    /* starting over a new block */
    endif
  endif
  $k \leftarrow k + 1$
  $done \leftarrow (l \geq m_1 \ or \ k \geq n)$
  endwhile
```

As soon as a *bad* column is encountered, the matrix being factored will have the appearance as shown below.



Since a back-solve and a re-triangularization is done every time a *bad* column is encountered, we need a column which is a full vector. But fortunately, we can just reserve one vector (whose size is equal to the number of rows) to store the intermediate computations. The typical structure of the matrix after the factorization is shown below.



The final upper trapezoidal form of C may have the following form.

$$\begin{pmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ 0 & C_{22} & C_{23} & C_{24} \\ 0 & 0 & C_{33} & C_{34} \end{pmatrix}$$

where $C_{ii}, i = 1,2,3$ have full row rank.

4.5 Issues in Parallel Implementation

The problem is to detect the rank of a large and sparse matrix C and obtain a QR factorization of that matrix. As was described in section 4.2, the general strategy is followed. We use SPARSPAK-B [40] for doing steps 1–3 as described in section 4.2. From the storage structure provided for R by SPARSPAK, we generate the static structure required for the factorization. Since this work involves only the symbolic structure of C

and is a well understood problem, we perform only the numerical factorization part on the parallel machine.

We consider issues in implementing this algorithm on a Hypercube architecture. In a rather straightforward way, the columns of the matrix C are wrapped around among the processors on the hypercube. For the sake of simplicity, we may assume that the processors form a ring, although for "broadcast" purposes, other connections of the hypercube are implicitly made use of. Each processor makes a decision as to include the next column in the factorization and sends a message to other processors along with the necessary transformations(if the column is included). The updating of the rest of the columns on the same processor is done only after sending the information to other processors.

There are a couple of obvious bottlenecks to this algorithm when implemented on a parallel machine. The "back-solve" process, when a *bad* column is found, involves accessing the partially formed upper triangular factor. Li and Coleman [61] discussed good back solve procedures for hypercubes that are effective on dense matrices. We used similar techniques in our implementation, but the over-all result is not impressive on sparse matrices because the arithmetic complexity is quite low compared to the communication overhead. This also causes the algorithm to come to a virtual pause, loosing some of the advantages of the asynchronous behavior of the algorithm. However, this happens only occasionally, so we can still expect some good speed-ups.

The "look-ahead" part of the algorithm, where it needs to find out which value of ρ is to be made permanent, is another bottleneck. There are a couple ways to get around this problem. We can make the look-ahead local to the columns held by that processor only. But then, we may be compromising on the quality of the estimate computed by the algorithm. The other alternative is to maintain a fixed number of possibilities (say $z = 4$) of the values of ρ and in the steady state, by the time we each processor is ready to process a column y, it would have enough information to fix the value of ρ corresponding to column $(y - z)$. We use the latter strategy in our implementation.

4.6 Implementation Details

We now give detailed account of the data structures used in implementing the algorithm. As has been noted before, the data structures are very similar to the ones that are used in sequential computations and hence we review them here for the sake of completeness.

4.6.1 Compressed and Uncompressed Subscript Notation

To represent a sparse matrix, we essentially need to keep track of two things — the value of each non-zero element as well as the position of each non-zero in the matrix. There are a lot of ways of achieving that goal but the following two schemes, originally due to Gustavson [48] and Sherman [71], are commonly used. Detailed descriptions may be found in George and Liu [39].

Suppose we are interested in storing an upper triangular matrix R, which is an $n \times n$ matrix. For the sake of illustration, we use the matrix R as shown in Figure 4.1. In the uncompressed format, we have a vector DIAG that stores the diagonal elements of R. The off-diagonal non-zero elements of R are stores by rows in a vector RNZ and an accompanying index vector XRNZ is used to point to the beginning of each row into the RNZ array. And the column indices of the off-diagonal non-zeros are stored in another vector NZSUB. It is easy to note that XRNZ also indexes into NZSUB vector to give us the start of each row. This scheme is illustrated in Figure 4.2.

In the compressed format, we still use DIAG, RNZ, and XRNZ. But now NZSUB holds the compressed subscripts and an accompanying index vector XNZSUB is used to point to the beginning of the subscript sequence for each row of R. Figure 4.3 illustrates this scheme.

The SPARSPAK-B, which we used to do the symbolic factorization, uses the compressed scheme for the upper triangular factor.

$$R = \begin{pmatrix} r_{11} & r_{12} & & r_{14} & & & \\ & r_{22} & & r_{24} & & & \\ & & r_{33} & & r_{35} & r_{36} & \\ & & & r_{44} & r_{45} & & \\ & & & & r_{55} & r_{56} & r_{57} \\ & & & & & r_{66} & r_{67} \\ & & & & & & r_{77} \end{pmatrix}$$

Figure 4.1: Example matrix R

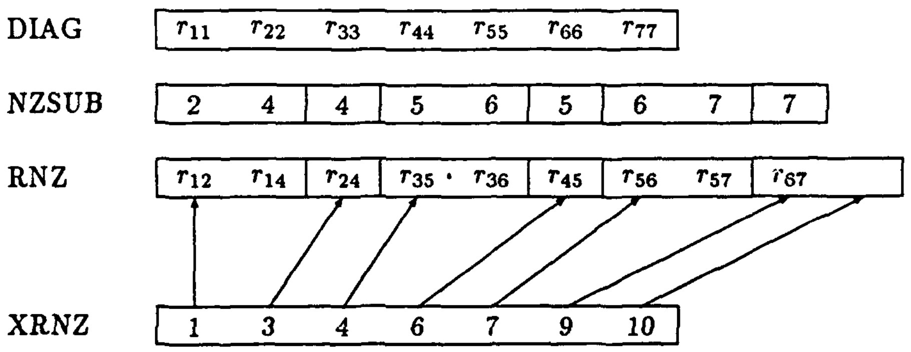

DIAG
r11 r22 r33 r44 r55 r66 r77
NZSUB
2 4 4 5 6 5 6 7 7
RNZ
r12 r14 r24 r35 r36 r45 r56 r57 r67
XRNZ
1 3 4 6 7 9 10

Figure 4.2: Uncompressed data storage scheme

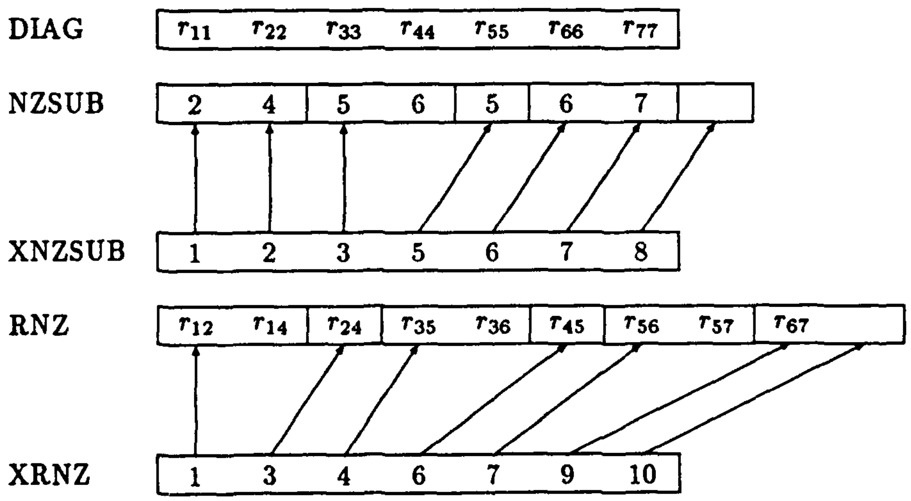

DIAG
r11 r22 r33 r44 r55 r66 r77
NZSUB
2 4 5 6 5 6 7
XNZSUB
1 2 3 5 6 7 8
RNZ
r12 r14 r24 r35 r36 r45 r56 r57 r67
XRNZ
1 3 4 6 7 9 10

Figure 4.3: Compressed data storage scheme

4.6.2 Static Data Structure for Rank Detection

We follow the first three steps of the George-Heath procedure, as described in section 4.2 initially. We used already available routines from SPARSPAK-A and SPARSPAK-B to do the same. But the matrix we use to give as the input is

$$G = \begin{pmatrix} A & 0 \\ C & I \end{pmatrix}$$

where I is an identity matrix of appropriate dimension so as to make the matrix G a square matrix. The reason for such a choice results from the following theorem from [41].

Theorem 4.1 *Let G be a square matrix of order n with a zero-free diagonal. Define Nonz(.) to be the non-zero structure of a sparse matrix and let L be the Cholesky factor of G^TG. Let G be reduced to upper triangular form using a sequence of Householder transformations and let G_k be the matrix G after the first k columns have been eliminated. Then*

$$Nonz(G_k) \subseteq Nonz(L + L^T) \qquad \text{for } k = 1, 2, \ldots, n-1.$$

Moreover, Q (the orthogonal matrix) can also be stored in the same static data structure.

It is to be noted however that this theorem heavily depends on the fact that the matrix F has a zero-free diagonal. But this assumption is required only at arriving at the static data structure and never used during the actual factorization. And hence we can introduce symbolic nonzeros for the purpose of arriving at the data structure and ignore them during the actual factorization.

Björck [16] provided the following result which tells us that the static data structure can be used for the factorization

Theorem 4.2 *Let $B = (b_1, b_2, \ldots \, b_n)$ be a column partition of B and let*

$$\hat{B} = (b_{j_1}, b_{j_2}, \ldots, b_{j_n}), \quad 1 \leq j_1 < j_2 \ldots < j_k \leq n$$

be a submatrix of B. Denote the Cholesky factors of $B^T B$ and $\hat{B}^T \hat{B}$ by R and $\hat{R}$ respectively. Then the nonzero structure of $\hat{R}$ is included in the nonzero structure of R.

In essence, we can skip any number of columns during the factorization and still use the static data structure as long as those columns are not reused for the factor.

4.7 Timing Results on a Hypercube

The above algorithm was implemented on an Intel iPSC/2 hypercube with 128 nodes, each with 4 megabytes of memory. The static data structure was generated on another machine (as that is not the part we are trying to parallelize) and was fed as input on the hypercube.

Since the data structure represented a symmetric matrix, it can be viewed as row or column oriented storage. However, the symmetric structure allows us to efficiently execute the steps of the form

$$for\ j \in Nonz(c^{[l]})$$

in our algorithm, wherein all the columns in which there is a nonzero entry in the l-th row need to be dealt with. Since this type of need arises in an inner loop of the algorithm, efficient implementation of that step is critical.

The constraint matrix used as input was generated randomly but the A was taken to be a tridiagonal matrix with diagonal elements as unity and off-diagonal nonzeros as 1.0E-3. The test matrix had 10000 columns with approximately 100000 nonzeros in the final factor. The results are tabulated in table 4.1. *The results indicate that each time the number of processors is doubled, the speed-up obtained is approximately equal to 1.4.*

4.8 Robustness of the Algorithm

To test the effectiveness of this estimator, we used this algorithm to estimate the condition number of a given matrix. As was suggested by Stewart [72], we generated

Table 4.1: Timing results on factorization with the condition estimator

no. of processors	time (secs)
2	40.59
4	28.59
8	20.42
16	14.69
32	10.35
64	7.34
128	5.24

random test matrices of dimension 10, 25 and 50 with a known condition number — the values being 1.0E1, 1.0E3, 1.0E6 and 1.0E9. For each of the possibilities, we generated two types of matrices — one where there is a sharp break in the singular value distribution and the other in which the singular values are exponentially distributed between 1 and the condition number.

The algorithm always estimated correctly (within 2 decimal digit accuracy), if there is a sharp break in the singular value distribution and hence the results in Table 4.2 only illustrate the case where there is an exponential distribution of singular values. For each dimension n, 50 test matrices were generated. k_2 is the actual condition number of the test matrix. The numbers quoted in each entry represent the minimum / average value of the ratio of the estimated condition number to the actual value. The results are rounded to two significant digits, so a ratio of 1.0 implies that the estimate had at least 2 correct digits.

Comparative results are included in Table 4.3 for LINPACK, and in Table 4.4 for Bischof's estimator.

Another way to see the effectiveness of the estimator is to estimate the rank of the matrix using the estimator during the factorization. The test matrices were generated as described above, but all of them have an exponential distribution of singular values.

Table 4.2: Our condition estimation tests

k_2	$n = 10$	25	50
10	0.36/0.67	0.33/0.53	0.30/0.43
10^3	0.20/0.58	0.20/0.42	0.22/0.37
10^6	0.11/0.48	0.12/0.36	0.10/0.27
10^9	0.12/0.51	0.12/0.33	0.09/0.26

Table 4.3: LINPACK condition estimation tests

k_2	$n = 10$	25	50
10	0.29/0.46	0.24/0.30	0.17/0.23
10^3	0.29/0.56	0.20/0.33	0.19/0.26
10^6	0.46/0.76	0.20/0.46	0.22/0.35
10^9	0.68/0.86	0.24/0.55	0.23/0.40

Table 4.4: Bischof's condition estimation tests

k_2	$n = 10$	25	50
10	0.56/0.77	0.59/0.71	0.63/0.71
10^3	0.33/0.53	0.40/0.50	0.31/0.45
10^6	0.12/0.53	0.16/0.38	0.24/0.36
10^9	0.16/0.45	0.17/0.33	0.19/0.31

Table 4.5: Rank detection tests of our algorithm (k_2 = 1.0e6)

n	min	avg	max
10	7/1.0e-4	7/1.0e-4	8/2.2 e-5
25	18/5.7e-5	19/3.2e-5	21/1.0e-5
50	38/3.0e-5	40/1.7e-5	43/7.2e-6

Comparative results with Bischof's estimator as well as column pivoting technique (for two values of the condition of the matrix $k_2 = 1.0E6$ and $1.0E9$) are tabulated in tables 4.5-4.10. In each of those tables, n is the size of the test matrix, k_2 is the condition number. Each entry is of the form j/s under each of the min / max / avg columns, where j is the rank detected by that algorithm and s is the j-th singular value of the test matrix. In all the estimations, we used a cut-off of $1.0E5$ to detect the rank, and hence this singular value gives us an indication of how good the estimate is. The following points summarize the test results.

- Our algorithm always gave a conservative estimate compared to the other two strategies.
- Column pivoting seems to obtain a stable value for the rank, with the smallest difference between its maximum and minimum estimates.
- All the algorithms appear to be reasonably accurate.

The following theorem, due to Barlow [11], confirms the first observation.

Theorem 4.3 *Let $rank_1(C)$ be the rank as determined by Algorithm 4.1 for a tolerance ϵ and let $rank_2(C)$ be the rank as determined by Algorithm 4.2 with the same tolerance. Then, excluding the effects of round-off errors, $rank_2(C) \leq rank_1(C)$.*

It is also to be noted that column pivoting does not share the property given in the above theorem [11].

Table 4.6: Rank detection tests of Bischof's algorithm (k_2 = 1.0e6)

n	min	avg	max
10	8/2.2 e-5	8/2.2e-5	9/4.6e-6
25	20/1.8e-5	21/1.0e-5	23/3.1e-6
50	41/1.2e-5	44/5.4e-6	47/2.3e-6

Table 4.7: Rank detection tests of Column Pivoting algorithm (k_2 = 1.0e6)

n	min	avg	max
10	8/2.2 e-5	8/2.2e-5	9/4.6e-6
25	21/1.0e-5	21/1.0e-5	22/5.6e-6
50	43/7.2e-6	43/7.2e-6	46/3.1e-6

Table 4.8: Rank detection tests of our algorithm (k_2 = 1.0e9)

n	min	avg	max
10	5/1.03-4	5/1.0e-4	6/1.0e-5
25	12/7.5e-5	13/3.2e-5	14/1.3e-5
50	24/6.0e-5	26/2.6e-5	28/1.1e-5

Table 4.9: Rank detection tests of Bischof's algorithm (k_2 = 1.0e9)

n	min	avg	max
10	5/1.03-4	5/1.0e-4	6/1.0e-5
25	12/7.5e-5	14/1.3e-5	15/5.6e-6
50	26/2.6e-5	28/1.1e-5	29/7.2e-6

Table 4.10: Rank detection tests of Column Pivoting algorithm (k_2 = 1.0e9)

n	min	avg	max
10	5/1.03-4	5/1.0e-4	6/1.0e-5
25	13/3.2e-5	13/3.2e-5	15/5.6e-6
50	27/1.7e-5	27/1.7e-5	29/7.2e-6

Chapter 5

Weighted Least Squares Problems

Consider the Equality Constrained Least Squares Problem (LSE), usually denoted by

$$\min_x \|b - Ax\|_2 \tag{5.1a}$$

subject to

$$Cx = d \tag{5.1b}$$

where $C \in \mathbf{R}^{m_1 \times n}$, $A \in \mathbf{R}^{m_2 \times n}$, $b \in \mathbf{R}^{m_2}$, $d \in \mathbf{R}^{m_1}$ with $m_1 \leq n \leq m_1 + m_2$. The problem LSE has an unique solution if and only if the matrix $B = \begin{pmatrix} A \\ C \end{pmatrix}$ has a rank of n [29].

Usually the matrix $B = \begin{pmatrix} A \\ C \end{pmatrix}$ is large and sparse. These problems arise in optimal design of structures [8], constrained optimization [30], geodetic least squares adjustments [14], and signal processing [32].

In this chapter, we consider several aspects in solving the above problem on a distributed memory multi-processors.

5.1 Traditional Methods

This problem has been well studied for sequential computations. Lawson and Hanson [59] discussed the following popular approaches.

5.1.1 Using the Basis of a Null Space

This method is due to Hanson and Lawson [50]. Let

$$X = \{x : Cx = d\}. \tag{5.2}$$

Then if

$$C = HRK^T \tag{5.3}$$

is any orthogonal decomposition of C and K is partitioned as

$$K = [K_1, K_2] \tag{5.4}$$

where $K_1 \in \mathbf{R}^{n \times k_1}$ and $K_2 \in \mathbf{R}^{n \times (n-k_1)}$, then

$$X = \{x : x = \bar{x} + K_2 y_2\} \tag{5.5}$$

where

$$\bar{x} = C^+ d \tag{5.6}$$

and y_2 ranges over the space of all vectors of dimension $(n - k_1)$. Then the minimization part reduces to finding a $(n - k_1)$-vector y_2 that minimizes

$$||A(\bar{x} + K_2 y_2) - b||_2 \tag{5.7}$$

or equivalently

$$||(AK_2)y_2 - (b - A\bar{x})||_2. \tag{5.8}$$

Hence the minimum length y_2 is given by

$$y_2 = (AK_2)^+(b - A\bar{x}) \tag{5.9}$$

and hence

$$x_{LSE} = C^+d + K_2(AK_2)^+(b - AC^+d). \tag{5.10}$$

This method is not suitable for sparse matrices because of excess fill-in created during the computation of (5.8). Moreover, to arrive at a static structure for the solution of (5.8) is not easy.

5.1.2 Direct Elimination

This method is due to Björck and Golub [18]. Suppose that the rank of C is m_1 and partition

$$\begin{pmatrix} C \\ A \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \\ A_1 & A_2 \end{pmatrix} \text{ and } x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \tag{5.11}$$

where $C_1 \in \mathbb{R}^{m_1 \times m_1}$ with $x_1 \in \mathbb{R}^{m_1}$. Then, assuming that C_1^{-1} exists, we have

$$x_1 = C_1^{-1}(d - C_2x_2) \tag{5.12}$$

and hence

$$\begin{aligned} \|Ax - b\|_2 &= \|A_1C_1^{-1}(d - C_2x_2) + A_2x_2 - b\|_2 \\ &= \|(A_2 - A_1C_1^{-1}C_2)x_2 - (b - A_1C_1^{-1}d)\|_2 \\ &= \|\tilde{E}_2x_2 - \tilde{f}\|_2 \end{aligned} \tag{5.13}$$

Suppose $C_1 = Q_1^T\tilde{C}_1$ is a QR decomposition of C_1. Then

$$\begin{aligned} \tilde{A}_2 &= A_2 - (A_1\tilde{C}_1^{-1})(Q_1C_2) \\ &= A_2 - \tilde{A}_1\tilde{C}_2 \end{aligned} \tag{5.14}$$

and

$$\begin{aligned}\tilde{b} &= b - (A_1\tilde{C}_1^{-1})(Q_1 d) \\ &= b - \tilde{A}_1\tilde{d}. \end{aligned} \tag{5.15}$$

Having solved the smaller dimensional problem, we finally compute

$$x_1 = C_1^{-1}(d - C_2 x_2). \tag{5.16}$$

Note that for sparse matrices this is not a good strategy since we can not predict the structure of the intermediate matrices. The step in equation (5.12) also creates a lot of fill-in for the intermediate matrices. A strategy based on this method due to Björck [17], is implemented in SPARSPAK-B.

5.1.3 Weighting

In this procedure, we first look at solving the following unconstrained problem

$$\min_{x \in \mathbf{R}^n} \left\| \begin{pmatrix} \tau C \\ A \end{pmatrix} x - \begin{pmatrix} \tau d \\ b \end{pmatrix} \right\|_2 \tag{5.17}$$

for some large, positive weight τ.

Let $x(\tau)$ be the solution to (5.17). By forming the normal equations to this problem, equation (5.17) is equivalent to

$$\begin{pmatrix} \tau^{-2} I_{m_1} & 0 & C \\ 0 & I_{m_2} & A \\ C^T & A^T & 0 \end{pmatrix} \begin{pmatrix} r_1 \\ r_2 \\ z \end{pmatrix} = \begin{pmatrix} d \\ b \\ 0 \end{pmatrix} \tag{5.18}$$

where I_{m_1} and I_{m_2} are identity matrices of size m_1 and m_2 respectively.

By considering the Lagrange multipliers [42] for equation (5.1), we can rewrite it as

$$\begin{pmatrix} 0 & 0 & C \\ 0 & I & A \\ C^T & A^T & 0 \end{pmatrix} \begin{pmatrix} \lambda \\ r \\ x \end{pmatrix} = \begin{pmatrix} d \\ b \\ 0 \end{pmatrix}. \tag{5.19}$$

It is easy to observe that from equation (5.18) and equation (5.19) above that

$$\lim_{\tau \to \infty} x(\tau) = x_{LSE} \tag{5.20}$$

provided that the $rank(C) = m_1$.

5.2 Deferred Correction

Powell and Reid [66] showed that for the weighting method, row ordering before the factorization and column ordering during factorization are very critical to the accuracy of the solution. In other words, some row ordering before the factorization and some sort of column pivoting during the factorization are to be used in practice. The row ordering does not affect the sparsity of the triangular factor, but the column ordering during the factorization is not *appropriate* for sparse matrices, especially on Distributed Memory machines. To overcome this problem, Van Loan [74] proposed a *Deferred Correction* scheme to obtain a better estimate from the original solution. The idea behind that scheme is outlined below.

Algorithm 5.1 *Deferred Correction*

1. *Choose τ and compute the solution $x(\tau)$ for equation (5.17)*

2. Set

$$x^{(1)} = x(\tau)$$
$$r^{(1)} = b - Ax(\tau)$$
$$\lambda^{(1)} = \tau^2(d - Cx(\tau))$$

3. For $k = 1, 2, \cdots,$

(a) Compute the residual

$$\begin{pmatrix} \delta_1^{(k)} \\ \delta_2^{(k)} \\ \delta_3^{(k)} \end{pmatrix} = \begin{pmatrix} d \\ b \\ 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 & C \\ 0 & I_m & A \\ C^T & A^T & 0 \end{pmatrix} \begin{pmatrix} \lambda^{(k)} \\ r^{(k)} \\ x^{(k)} \end{pmatrix}$$

(b) Solve

$$\begin{pmatrix} \tau^{-2} I_P & 0 & C \\ 0 & I_m & A \\ C^T & A^T & 0 \end{pmatrix} \begin{pmatrix} \Delta\lambda^{(k)} \\ \Delta r^{(k)} \\ \Delta x^{(k)} \end{pmatrix} = \begin{pmatrix} \delta_1^{(k)} \\ \delta_2^{(k)} \\ \delta_3^{(k)} \end{pmatrix}$$

(c) Set

$$\lambda^{(k+1)} = \lambda^{(k)} + \Delta\lambda^{(k)}$$
$$r^{(k+1)} = r^{(k)} + \Delta r^{(k)}$$
$$x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$$

Barlow [4, 5] gave an efficient and robust implementation of the deferred correction procedure along with some results on convergence of the algorithm. He also showed that choosing $\tau = \mu^{-\frac{1}{3}}$ gives a solution as good as can be expected in *two* iterations, for all but ill-conditioned problems. and that a column pivoting strategy based entirely on the matrix C can be used to factor the matrix B in a stable manner. We describe the modified procedure below.

Algorithm 5.2 *Modified Van Loan's Procedure*

1. Set

$$x^{(1)} = x(\tau) \textit{ where } x(\tau) \textit{ is the solution of equation (5.17)}$$

$$r^{(1)} = r(\tau) = b - Ax(\tau)$$

$$w_1^{(1)} = d - Cx(\tau)$$

$$\lambda^{(1)} = \tau^2 w_1^{(1)}$$

2. For $k = 1, 2, \cdots,$

(a) Solve the system

$$\min_{\Delta x^{(k)}} \left\| \begin{pmatrix} \tau C \\ A \end{pmatrix} \Delta x^{(k)} - \begin{pmatrix} \tau w_1^{(k)} + \tau^{-1}\lambda^{(k)} \\ r^{(k)} \end{pmatrix} \right\|_2 \tag{5.21}$$

(b) set

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + \Delta x^{(k)} \\ r^{(k+1)} &= r^{(k)} - A\Delta x^{(k)} \\ w_1^{(k+1)} &= w_1^{(k)} - C\Delta x^{(k)} \\ \lambda^{(k+1)} &= \lambda^{(k)} + \Delta x^{(k)} = \tau^2 w_1^{(k+1)} \end{aligned}$$

5.3 Proper Choice for Parallel Implementation

Of the three methods described above, the weighting approach is well suited for parallel implementations because of its simplicity. We are essentially solving unconstrained least squares problem. However as was shown by Van Loan [74] and Barlow [4], care must be taken to see that the rank of C is detected properly and deferred correction be applied to improve the solution. We already examined an incremental condition estimation technique in the previous chapter, that allows us to factor a matrix using a static data structure, yet detect the rank quite accurately.

Moreover, we can not but over-emphasize the view that on distributed memory multiprocessors, maintaining dynamic data structures for representing the sparse matrix is not efficient. Firstly, there is no rational basis on which the required data structures could be distributed over all the nodes. That means that all the data structures are to be held on all the nodes. And as a consequence, any change in the sparsity structure calls for updating all the copies held by all the nodes and we know of no way how this can be done efficiently. In this context the deferred correction approach outweights the other two approaches.

5.4 Issues in Parallel Implementation

Once the matrix $\begin{pmatrix} \tau C \\ A \end{pmatrix}$ is factored, all the steps in the algorithm are either a matrix-vector multiplication or a forward solve or a backward solve, except the solution of equation (5.21). There are a couple of ways to solve the equation (5.21).

1. If the orthogonal factor Q is available in factored form, then the solution of equation (5.21) is quite easy. We just need to apply the orthogonal factor to the new right hand side expression every time and perform a back solve. Our static data structure, as described in section 4.6 is capable of storing Q in factored form. However, the Q would be distributed over all the nodes and applying Q to the right hand side expression would be highly sequential in nature to be executed on a parallel machine.

2. The other way is to use semi-normal equations, as described by Björck [15]. The basic idea here is as follows. Suppose we want to solve

$$\min_y \|h - Gy\|_2 \tag{5.22}$$

then, under the normal equations approach, we would be solving

$$G^T G y = G^T h. \tag{5.23}$$

Suppose we have an orthogonal factorization of G, such as

$$G = QR,$$

then it is easy to see that $R^T R = G^T G$ and so we could solve

$$R^T R y = G^T h \tag{5.24}$$

instead of solving equation (5.23) to get a solution. Note that solving equation (5.24) involves a matrix-vector multiplication followed by a forward solve and backward solve only. Björck showed that if one step of iterative refinement is done by solving

$$R^T R \Delta y = G^T (h - Gy) \tag{5.25}$$

then $y + \Delta y$ is as good a solution as can be expected under most of the circumstances for the problem (5.22).

We have chosen to implement the second method here, even though it involves two back solves and two forward solves per each iteration, instead of storing Q, as shown below. We first solve

$$R^T R \Delta \tilde{x}^{(k)} = (\tau C^T \;\; A^T) \begin{pmatrix} \tau w_1^{(k)} + \tau^{-1} \lambda^{(k)} \\ r^{(k)} \end{pmatrix} \tag{5.26}$$

and perform an iterative improvement step

$$R^T R \Delta q^{(k)} = (\tau C^T \;\; A^T) \left\{ \begin{pmatrix} \tau w_1^{(k)} + \tau^{-1} \lambda^{(k)} \\ r^{(k)} \end{pmatrix} - \begin{pmatrix} \tau C \\ A \end{pmatrix} \Delta \tilde{x}^{(k)} \right\} \tag{5.27}$$

and set

$$\Delta x^{(k)} = \Delta \tilde{x}^{(k)} + \Delta q^{(k)}. \tag{5.28}$$

Table 5.1: Timing results of constrained problem

No. of Processors	time in secs.
2	105.16
4	76.76
8	57.28
16	42.12
32	31.20
64	22.94
128	17.12

5.5 Tests on Parallel Implementation

The constraint matrix C is taken to be of random sparsity structure with random elements values and the matrix A is taken as a tridiagonal matrix with unit diagonal elements and the off-diagonal values being 1.0E-3.

As was described in previous chapter, we generated a static data structure for carrying out the factorization of the constraint matrix. Once the rank of C and the columns that re included in the factorization are determined, we recompute the R factor, but this time the matrix we factor is $\begin{pmatrix} \tau C \\ A \end{pmatrix}$. The important point to be noted is that the same static data structure is capable of holding the new factor R. By the updating the right-hand side simultaneously during the factorization, we can get the initial solution $x(\tau)$ by doing a simple back solve. And we perform two steps of deferred correction procedure, as described in previous section.

Test matrices up to the size of 10,000 columns were generated and up to 128 processors of the hypercube are used to determine the speed-up. The results are tabulated in table 5.1. Once again, we note that every time the number of processors is doubled, we get an effective speed-up of about 1.35. The drop in performance can be attributed to the back solve processes, especially during the deferred correction procedures.

Table 5.2: Trace of the Deferred Correction Algorithm

Iteration	$\|\Delta x\|_2$	$\frac{\|\Delta x\|_2}{\|x\|_2}$	$\|w_1\|_2 + \|w_3\|_2$
1	0.29E-13	0.23E-13	0.22E-13
2	0.90E-14	0.70E-14	0.54E-14
3	0.41E-14	0.32E-14	0.32E-14
4	0.27E-14	0.21E-14	0.27E-14
5	0.24E-14	0.19E-14	0.25E-14

5.6 Effectiveness of Deferred Correction

To show that the modified deferred correction works well in practice on sparse matrices, we generated sparse matrices with random elements as well as random sparsity structure, with the number of non-zeros in each row being in a range of a small fraction of the number of columns. This matrix is taken as the constraint matrix. The matrix A was taken to be a tridiagonal matrix with unit diagonal and the off-diagonal elements being 1.0E-03. The right-hand side was also taken to be a vector of unit elements. 100 sparse matrices up to the size of 2,000 columns were tested on a SUN-4 architecture in double precision. The value of τ was chosen to be equal to $\mu^{\frac{-1}{3}}$ where μ is the machine precision.

In all cases, the solution converged in two iterations, the convergence criterion being $\|w_1^{(1)}\|_2 + \|w_3^{(k)}\|_2 \leq 10^{-12}$. Here $w_3^{(k)} = -C^T\lambda^{(k)} - A^T r^{(k)}$. The worst-case progress of the algorithm on a random matrix is shown in table 5.2. Barlow [4,5] shows evidence that two iterations give us a solution that is as good as can be expected, for all but ill-conditioned problems.

5.7 Conclusions

We examined three algorithms for computing an orthogonal factorization of a dense matrix on distributed memory multi-processors. The factorization is computed as a sequence of given's rotations because there is a lot of freedom in the order in which those rotations are applied. The knight's tour givens ordering works on any ring connected architecture and was shown to be not optimal in terms of computational complexity. The greedy givens algorithm exploits the hypercube connections during the recursive elimination phase and is easy to implement. It seems to be asymptotically optimal in terms of the communication complexity. The recursive fine partition algorithm also makes use of the hypercube connections but involves more communication than the greedy algorithm. However it keeps all the processors busy most of the time.

We then consider rank detection of a sparse matrix during factorization, especially when the factorization is performed on a distributed memory multi-processor. We first emphasize the use of static data structures to do the sparse factorization. We describe a rank detection algorithm using an incremental condition estimator that is quite effective for parallel sparse matrix factorizations. We show that static structures can be used to carry out the factorization. We show experimental results that suggest that the rank detection strategy is comparable to that of the column pivoting and a recent algorithm by Bischof.

Lastly we consider solving equality constrained least squares problems on a distributed memory multi-processor. After examining popular approaches for solving this problem, we conclude that the weighted least squares approach is the most efficient way to solve large and sparse instances. A major concern in solving these problems is the accurate rank detection of the constraint matrix. We make use of a procedure previously described to detect the rank. We also note that a static structure can be generated that can be used for the rank detection of the constraint matrix as well as the rest of the solution process. To improve the accuracy of the solution from the weighted least squares approach, we perform two steps of modified deferred correction technique . We show evidence that the solution after two steps of deferred correction is as good as can be expected, for all but ill-conditioned problems. We also show good speed-ups in solving large and sparse equality conditioned least squares problems on hypercubes up to 128 processors.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Compueter Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[2] Gita Alaghband. Parallel pivoting combined with parallel reduction. Technical Report ICASE 87-75, NASA Langley Research Center, Hampton, VA, 1987.

[3] Gene M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, AFIPS Press, Reston, VA, April 1967.

[4] Jesse L. Barlow. Error analysis and implementation aspects of deferred correction for equality constrained least squares problems. *SIAM Journal on Numerical Analysis*, 25(6):1340–1358, December 1988.

[5] Jesse L. Barlow. Convergence of deferred correction for equality constrained least squares problems in two iterations. in preparation, 1990.

[6] Jesse L. Barlow and Susan L. Handy. The direct solution of weighted and equality constrained least squares problems. *SIAM Journal on Scientific and Statistical Computing*, 9(4):704–16, July 1988.

[7] Jesse L. Barlow and I. C. F. Ipsen. Parallel scaled Givens rotations for the solution of linear least squares problems on a systolic array. *SIAM Journal on Scientific and Statistical Computing*, 8:716–733, September 1987.

[8] Jesse L. Barlow, N. K. Nichols, and Robert J. Plemmons. Iterative methods for equality constrained least squares problems. *SIAM Journal on Scientific and Statistical Computing*, 9(5):892–906, September 1988.

[9] Jesse L. Barlow and Udaya B. Vemulapati. An improved method for one-way dissection with singular diagonal blocks. *SIAM Journal on Matrix Analysis and Applications*, 11(4), October 1990.

[10] Jesse L. Barlow and Udaya B. Vemulapati. Incremental condition estimator for parallel sparse matrix factorizations. In *Procedings of the Fifth Distributed Memory and Concurrent Computer Conference*. IEEE Publications, April 1990.

[11] Jesse L. Barlow and Udaya B. Vemulapati. Rank detection. in preparation, 1990.

[12] Christan H. Bischof. Incremental condition estimation. *SIAM Journal on Matrix Analysis and Applications*, 11(2):312–322, April 1990.

[13] Christan H. Bischof, John G. Lewis, and Daniel J. Pierce. Incremental condition estimation for general matrices. Technical Report MCS-P106-0989, Mathematics and Computer sciences division, Argonne National Laboratory, Argonne, IL, September 1989.

[14] Åke Björck. A general updating algorithm for constrained linear least squares problems. *SIAM Journal on Scientific and Statistical Computing*, 5:394–402, 1984.

[15] Åke Björck. Stability analysis of the method of semi-normal equations for linear least squares problems. *Linear Algebra and Its Applications*, 88/89:31–48, April 1987.

[16] Åke Björck. A direct method for sparse least squares problems with lower and upper bounds. *Numerische Mathematik*, 54:19–32, 1988.

[17] Åke Björck and Iain S. Duff. A direct method for the solution of sparse linear least squares problems. *Linear Algebra and Its Applications*, 34:43–67, 1980.

[18] Åke Björck and Gene H. Golub. Iterative refinement of linear least squares solutions by Householder transformations. *BIT*, 7:322–337, 1967.

[19] R. M. Chamberlain and M. J. D. Powell. QR factorization for linear least squares problems on a hypercube multiprocessor. *IMA Journal of Numerical Analysis*, 8(4):401–413, 1989.

[20] Tony F. Chan. Rank revealing QR factorizations. *Linear Algebra and Its Applications*, 88/89:67–82, 1987.

[21] Eleanor Chu and Alan George. Sparse orthogonal decomposition on a hypercube multiprocessor. *SIAM Journal on Matrix Analysis and Applications*, 11(3):453–465, July 1990.

[22] A. K. Cline, A. R. Conn, and Charles F. Van Loan. *Generalizing the LINPACK condition estimator*, volume 909 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1982.

[23] Thomas F. Coleman, Anders Edenbrandt, and John R. Gilbert. Predicting fill for sparse orthogonal factorization. *Journal of the ACM*, 33(3):517–532, 1986.

[24] M. Cosnard, J. M. Muller, and Y. Robert. Parallel QR decomposition of a rectangular matrix. *Numerische Mathematik*, 48:239–249, 1986.

[25] George Cybenko, David W. Krumme, and K. N. Venkataraman. Fixed hypercube embedding. *Information Processing Letters*, 25(1):35–39, 1987.

[26] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, 1979.

[27] Iain S. Duff. Private Communication, 1990.

[28] Editor. Editor's note. *SIAM Journal on Scientific and Statistical Computing*, 9(4):608, July 1988.

[29] L. Elden. Perturbation theory for the least squares problem with equality constraints. *SIAM Journal on Numerical Analysis*, 17:338–350, 1980.

[30] R. Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, Second edition, 1987.

[31] L. V. Foster. Rank and null space calculations using matrix decomposition without column pivoting. *Linear Algebra and Its Applications*, 74:47–72, 1986.

[32] O. L. Frost. An algorithm for linearly constrained adaptive array processing. *Proc. IEEE*, 60:926–935, 1972.

[33] George A. Geist and Charles H. Romine. LU factorization algorithms on distributed memory multiprocessor architectures. *SIAM Journal on Scientific and Statistical Computing*, 9(4):639–649, July 1988.

[34] W. M. Gentleman. Error analysis of QR decomposition by Givens transformations. *Linear Algebra and Its Applications*, 10:189–197, 1975.

[35] W. M. Gentleman and H. T. Kung. Matrix triangularization by systolic arrays. *Proc. SPIE Symposium*, 298:19–26, August 1981.

[36] Alan George and Michael T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and Its Applications*, 34:69–83, 1980.

[37] Alan George, Michael T. Heath, Joseph W. H. Liu, and Esmond Ng. Sparse Cholesky factorization on a local memory Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9(2):327–340, March 1988.

[38] Alan George and Joseph W. H. Liu. An optimal algorithm for symbolic factorization of symmetric matrices. *SIAM Journal on Computing*, 9:583–593, 1980.

[39] Alan George and Joseph W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, Englewood Cliffs, N. J., 1981.

[40] Alan George and Esmond Ng. SPARSPAK: Waterloo sparse matrix package user's guide for SPARSPAK-B. Technical Report CS-84-37, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, November 1984.

[41] Alan George and Esmond Ng. Orthogonal reduction of sparse matrices to upper triangular form using Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 7(2):460–472, April 1986.

[42] Phillip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization.* Academic Press, New York, 1981.

[43] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins Press, Baltimore, second edition, 1989.

[44] M. Gunzberger and R. Nicholaides. Elimination with noninvertible pivots. *Linear Algebra and Its Applications*, 64:183–189, 1985.

[45] M. Gunzberger and R. Nicholaides. On substructuring algorithms and solution techniques for the numerical approximation of partial differential equations. *Applied Numerical Mathematics*, 64:243–256, 1986.

[46] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[47] John L. Gustafson, G. R. Montry, and R. E. Brenner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, July 1988. Winner of Gordon Bell Award.

[48] F. G. Gustavson. Some basic techniques for solving sparse systems of equations. In D. J. Rose and R. A. Willoughby, editors, *Sparse Matrices and their Applications*, pages 41–52. Plenum Press, New York, 1972.

[49] W. W. Hager. Condition estimators. *SIAM Journal on Scientific and Statistical Computing*, 5:311–316, 1984.

[50] Richard J. Hanson and Charles L. Lawson. Extensions and applications of the Householder algorithm for solving linear least squares problems. *Math. Comp.*, 23(108):787–812, 1969.

[51] Michael T. Heath. Some extensions of an algorithm for sparse linear least squares problems. *SIAM Journal on Scientific and Statistical Computing*, 3:223–237, 1982.

[52] Michael T. Heath and Charles H. Romine. Parallel solution of triangular systems on distributed memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9(3):558–588, May 1988.

[53] Don E. Heller and I. C. F. Ipsen. Systolic networks for orthogonal decomposition. *SIAM Journal on Scientific and Statistical Computing*, 4:261–269, 1983.

[54] Nicholas J. Higham. A survey of condition number estimation for triangular matrices. *SIAM Review*, 29(4):575–596, 1987.

[55] Lennart S. Johnson and Ching Tien Ho. Algorithms for matrix transposition on boolean N-cube configured ensemble architectures. *SIAM Journal on Numerical Analysis*, 9(3):419–454, 1989.

[56] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[57] Vijay K.Naik and Merrell L. Patrick. Communication requirements of sparse Cholesky factorization with nested dissection ordering. In *Parallel Processing for Scienctific Computing*, pages 9–14. SIAM Publications, Philadelphia, PA, 1987.

[58] David W. Krumme, K. N. Venkataraman, and George Cybenko. Hypercube embedding is NP-Complete. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986*. SIAM Publications, 1986.

[59] Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems*. Prentice Hall, Englewood Cliff, N.J., 1974.

[60] Guangye Li and Thomas F. Coleman. A parallel triangular solver for distributed memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9(3):485–502, May 1988.

[61] Guangye Li and Thomas F. Coleman. A new method for solving triangular systems on distributed memory message-passing multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 10(2):382–396, March 1989.

[62] Oliver A. McBryan and Eric F. Van De Velde. Hypercube algorithms and implementations. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s227–s287, March 1987.

[63] J. J. Modi and M. R. B. Clarke. An alternative Givens ordering. *Numerische Mathematik*, 43:83–90, 1984.

[64] Alex Pothen. *Sparse Null Bases and Marriage Theorems*. PhD thesis, Cornell University, Ithaca,NY, 1984.

[65] Alex Pothen, Somesh Jha, and Udaya Vemulapati. Orthogonal factorization on the hypercube. In *Hypercube Multiprocessors 1987*. SIAM Publications, 1987.

[66] M. J. D. Powell and J. K. Reid. On applying Householder transformations to linear least squares problems. *Proc. IFIP Congress*, pages 122–6, 1968.

[67] Micheal J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, Inc., 1987.

[68] Donald J. Rose, Robert E. Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5:266–283, 1976.

[69] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *Journal of the ACM*, 25(1):81–91, 1978.

[70] David S. Scott, Enrique Castro-Leon, and Edward J. Kushner. Solving very large dense system of linear equations on the iPSC/860. In *Proceeding of the Fifth Distributed Memory Compting Conference.* IEEE Computer Society Press, 1990.

[71] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28:22–33, 1985.

[72] Andrew H. Sherman. On the efficient solution of sparse systems of linear ans nonlinear equations. Research Report 46, Dept. of Computer Science, Yale University, New Haven, CT, 1975.

[73] G. W. Stewart. The efficient generation of random orthogonal matrices with an application to condition estimators. *SIAM Journal on Numerical Analysis*, 17:403–9, 1980.

[74] Charles F. Van Loan. On the method of weighting for equality-constrained least-squares problems. *SIAM Journal on Numerical Analysis*, 22:851–864, 1985.

[75] Charles F. Van Loan. On estimating the condition of eigenvalues and eigenvectors. *Linear Algebra and Its Applications*, 88/89:715–732, 1987.

[76] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, London, 1965.