EFFICIENT PORTABLE PARALLEL MATRIX COMPUTATIONS

James Walter Juszczak

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188

TR-89-38 December 1989

Abstract

In this thesis we exercise a method of developing parallel algorithms for matrix computations that facilitates efficient and portable implementations. The method includes defining a set of communication primitives, selecting a storage scheme, embedding a logical communications topology in the physical architecture, and synchronizing data flow and computations to reduce the overhead of communications. Several algorithms are implemented using the column wrapped storage scheme and communication primitives which are independent of the underlying parallel architecture. Theoretical and experimental results are presented.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The objective of this thesis is to present, analyze and test a method of developing parallel matrix algorithms. The method involves defining a set of communication primitives, selecting a storage scheme, embedding a logical communications topology in the physical architecture, and synchronizing data flow and computations so as to reduce the overhead of communications. A consequence of this work is the development of portable and efficient code to perform a variety of matrix computations in parallel. It is envisioned that ultimately a package of parallel linear algebra routines (PLAPACK) will emerge from this effort.

The recent focus on parallel computing has been in response to the need for higher performance computers. Problems have been encountered that require intensive computations, which cannot be completed in a reasonable amount of time on the current generation of computers. The options are to construct faster hardware or use off the shelf technology to build arrays of processors and perform computation in parallel. We will explore how to exploit the latter of these options in this thesis.

We begin by defining some terminology which we will use to describe the implementation and evaluation of parallel algorithms. Since the objective of parallel computation is to execute algorithms in less time, we will define a measurement by which we can compare algorithms. *Speedup* is the ratio of the time taken by a computer to execute an equivalent serial algorithm and the time taken by the same computer to execute the parallel algorithm using p processors [1]. *Efficiency* is the speedup divided by the number of processors. The efficiency provides a measure of the performance cost and will indicate whether an algorithm fully utilizes the

[1]Speedup is sometimes defined as the ratio of the times taken by the fastest serial algorithm over the parallel algorithm.

available processors. Our goal is to approach linear speedup (i.e., speedup $= p$) and 100 per cent efficiency.

1.1 Interprocessor Communications

In this paper we will concentrate on MIMD (**m**ultiple **d**ata stream **m**ultiple **i**nstruction stream) machines with distributed memory. Each processor will operate in an asynchronous manner executing instructions and operating on data stored in its local memory. Each processor will have access to only its local memory and all communication and synchronization will be done by message passing.

Messages are passed through explicit calls to the communication library. This library contains several routines by which processors can send or receive messages. These primitives have been postulated as sufficient to implement a wide range of matrix algorithms [4, 14]. These are:

- node-to-neighbor
- broadcast
- total exchange
- data transpose
- one-way-shift
- vector sum
- inner product
- global compare

By restricting communications to this library of primitives it is possible to execute the calling routines on any machine on which these primitives have been implemented. It is through this communications library that we attain portability of the code. Moreover, by optimizing their implementation on a specific architecture we are able to take advantage of the particular network and achieve efficiency as well as portability. The communications primitives will be discussed in more detail in Chapter 2.

$$\begin{array}{cccccc}
\mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times
\end{array}$$

Figure 1.1: Column Wrapped Storage Scheme for $p = 3$ and $n = 6$

1.2 Static Load Balancing

Having decided on a set of communication primitives, we must concern ourselves with the distribution of the problem among the processors in order to evenly balance the work load, maximizing parallel activity while minimizing communications. Assuming a homogeneous system, we can attain higher efficiencies by assigning equal amounts of work to each processor.

For each of the algorithms in this paper we employ the *column wrapped* storage scheme, depicted in Figure 1.1, to store the matrix in the distributed memories [3, 5, 6, 7]. Given an indexed ordering of p processors $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$, this scheme assigns the ith column ($i = 1, 2, \ldots, n$) to processor $\mathbf{P}_{(i-1)\bmod p}$. Therefore, the first column is assigned to $\mathbf{P}_0$, the pth column to $\mathbf{P}_{p-1}$, the $(p+1)$th column to $\mathbf{P}_0$ and so on. If the number of processors divides the number of columns, n, then each processor receives an equal share of the matrix and presumably an equal share of the work. Otherwise ($n \bmod p$) processors will have $\lceil n/p \rceil$ columns and the rest will have $\lfloor n/p \rfloor$ columns. For $n \gg p$ the imbalance is insignificant. One advantage of the column wrapped storage scheme is that it is simple; since contiguous blocks (columns) of the matrix have been extracted, accessing elements will be straightforward involving little indexing overhead. Note, that for the column wrapped storage scheme, p cannot

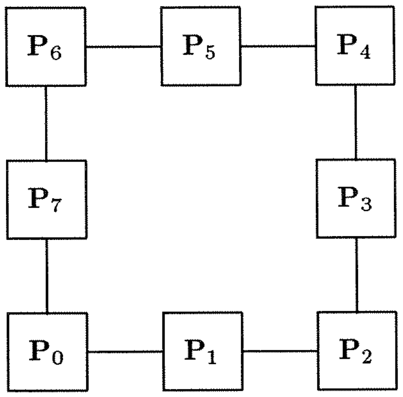

Figure 1.2: Ring of Processors

exceed n.

1.3 Architectures - ring, mesh, hypercube

In this section we will discuss three common distributed memory architectures: the ring, the mesh and the hypercube. In each of these models, it is possible to embed a logical ring of processors in the physical architecture so that neighbors in the ring are also neighbors in the underlying connection network. We will see that when an embedded ring is combined with the column wrapped storage scheme, high efficiencies can be obtained. This is advisable when considering some matrix algorithms since computations can be viewed as progressing across the matrix column by column.

The ring multiprocessor consists of p processors, $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$, connected in a ring as shown in Figure 1.2, where nodes P_i and P_j are neighbors if $(i+1) \bmod p = j$ (or $(j+1) \bmod p = i$).

It is assumed that each processor can simultaneously send to both neighbors or simultaneously send to one neighbor and receive from the other neighbor. In this thesis we assume that a ring can be embedded in the underlying physical architecture.

In a mesh network the processors are arranged in a d-dimensional lattice.

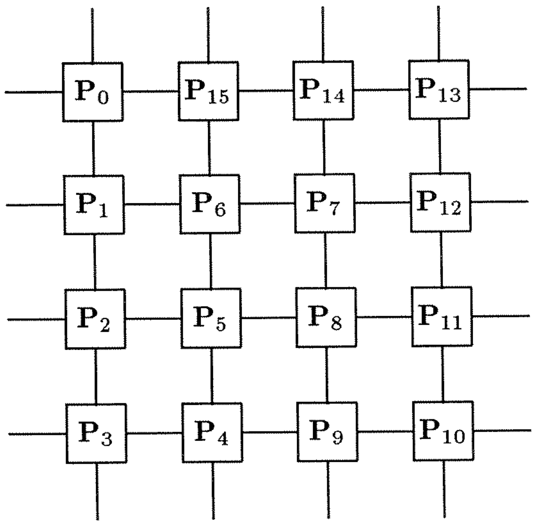

Figure 1.3: Two Dimensional Mesh Network with Embedded Ring

An interior processor has two neighbors in each dimension, giving it $2d$ neighbors. If the mesh has wrap-around connections then every processor has $2d$ neighbors. A processor can only communicate directly with these neighbors. Figure 1.3 shows a two dimensional mesh in which a ring has been embedded. Processors on the edges of the grid have wrap around connections to processors on the opposite edge.

A hypercube is a loosely coupled multiprocessor based on a binary n-cube network. An n dimensional hypercube has 2^n processors and each processor has n neighbors. The processors can be numbered $\mathbf{P}_0, \ldots, \mathbf{P}_{2^n-1}$, so that neighboring processors differ in exactly one place of the binary representation of their index. Figure 1.4 shows 1, 2 and 3 dimensional hypercubes. The following sequence of processor indices describes an embedded ring in a 3 dimensional hypercube, 000, 010, 011, 001, 101, 111, 110, 100 .

By restricting ourselves to a ring, the simplest and least efficient architecture, we examine the worst case scenario.

1.3.1 SYMULT S2010

All algorithms discussed in this thesis were implemented and tested on the SYMULT Series 2010 parallel processor belonging to the Computer Sciences Department of The University of Texas at Austin. The S2010 is a multiple data multiple

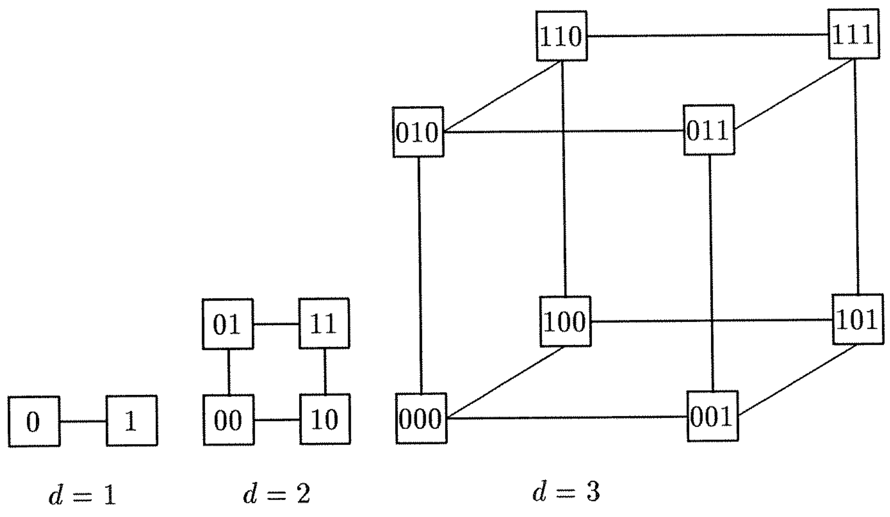

Figure 1.4: Hypercubes with Dimensions 1, 2, and 3

instruction parallel processing system. This machine has twenty-four Motorola 68020 microprocessors connected via automatic message routing device(AMRD) circuits in a 6×4 array topology. Each AMRD has five channels, four for communications to other AMRDs and one for access to its own local memory. Each AMRD can send up to 20 MBytes per second on any two channels simultaneously. When allocating a set of nodes on which to run an application a ring of an even number of nodes can be explicitly requested but owing to the AMRD circuits this is not necessary to obtain good results. Eight of the processors have 3 Mbytes of local memory and the rest each have 1 Mbyte. Applications utilizing more than 8 nodes were restricted in size because of the limited memory on 16 nodes.

All algorithms were implemented using double precision computer arithmetic in the C programming language on a SUN III (UNIX host), under a runtime system called the *Cosmic Environment*, which interacts with a node operating system running and on the Symult S2010 multicomputer called the *Reactive Kernel.* The host and node programs make procedure or function calls to routines from the set of communication primitives discussed in Chapter 2, and a set of BLAS routines written in C and optimized for the Motorola 68020 microprocessor [2].

[2]courtesy of Jim Meyering at the University of Texas at Austin

When comparing the expected and observed timings of the algorithms, it is necessary first to measure the *floprate* or speed at which a node of the multicomputer performs a computation. We will abide by convention and define a *flop* as the operation

$$y \leftarrow \alpha x + y \qquad\qquad \alpha \in \mathbf{R};\ x, y \in \mathbf{R}^n$$

in which two floating point operations are performed [2]. The flop rate of a single node of the SYMULT was timed through repeated calls to the BLAS routine, *saxpy*, on vectors of length 100, and a flop rate of 71 Kflops was attained. The time for a single flop is therefore, $\gamma = .000014$ seconds.

The cost of a communication between two processors is expressed as $\alpha + m\beta$ where α is the communication startup time, β is the per item (double precision floating point word) transmission time and m is the number of items being transmitted. The startup and transfer times were measured on the SYMULT S2010 and found to be: $\alpha \approx 450\mu$sec and $\beta \approx 10\mu$sec/item [15]. These measurements are with respect to the communication primitives (i.e., they include buffer manipulations in addition to those performed by the Cosmic Environment [10]).

1.4 Notation

The routines which perform the actual matrix computations will be discussed in their respective chapters. In doing so, we present the algorithms in a systematic way, using consistent notation discussed in this section.

The naming conventions and calling sequences of the LINPACK library developed at the Argonne National Laboratory were adopted where possible to conform to the LINPACK template.

When discussing the algorithms and the theory supporting them, the following notational conventions were used. The set of real numbers is denoted as $\mathbf{R}$. $\mathbf{R}^n$ is the set of all n-dimensional vectors with real elements and $\mathbf{R}^{m \times n}$ is the set of all $m \times n$ real matrices. Following Stewart [11], lower case Greek letters are used to represent scalars and lowercase Latin letters represent vectors. Uppercase Latin letters denote matrices. Elements of a vector or matrix are denoted with the Greek letter that best corresponds to the Latin letter identifying that vector or matrix. For example, α_{ij} is the element in the ith row and jth column of the matrix A and ξ_i is the ith component of the vector x. The letters n, m, i, j, k are reserved to represent integer dimensions and indices into vectors or matrices. The index i will hereafter be reserved to index processors in the ring. The superscript T is used to denote the transpose of a vector or matrix (e.g., $y = (v_1, v_2, \ldots, v_n)^T$ denotes a column vector).

Superscripts appearing in parentheses indicate the iteration number during which the variable exists. $A^{(k)} = (a_1^{(k)} \ldots a_n^{(k)})$ is a column partitioning of $A^{(k)}$. In a ring of p nodes, $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$, node $\mathbf{P}_i$ has column k if $(k-1) \bmod p = i$. We denote this as $k \in \mathbf{P}_i$ and this processor is also referred to as $\mathbf{P}(k)$. Therefore the symbols $\mathbf{P}_i$ and $\mathbf{P}(k)$ serve double duty, representing either a processor or a set of indices. We hope that each instance of their use is made clear by the context in which they appear.

1.5 Overview

The intent of this thesis is to demonstrate to the reader through a few representative examples that an effective method of parallelizing matrix algorithms is being presented.

In Chapter 2, we examine in detail the problem of exchanging data in a ring of processors. We will describe algorithms to perform a set of data exchange operations considered sufficient to perform matrix computations. For those operations used in this thesis, we will derive the time complexity of the algorithm as it has been implemented for a ring of processors.

Chapter 3 is concerned with the solution of systems of linear equations. The three algorithms presented in this chapter are : Gaussian elimination with partial pivoting, Cholesky decomposition and triangular solve. The first two algorithms factor a matrix into the product of simpler matrices thereby simplifying subsequent calculations involving the matrix. Both of these algorithms are parallelized using only the broadcast primitive. The triangular solve algorithm receives the factored matrix and the right hand side of the equation, $Ax = b$, and solves for the vector x. The parallel version of this algorithm employs the node-to-neighbor communication primitive.

Chapter 4 is concerned with the QR-factorization of matrices which can be applied to solving the linear least squares problem. In this chapter, the QR decomposition by Householder transformations and the Modified Gram Schmidt algorithms are both implemented in parallel using only the broadcast primitive.

In Chapter 5, we discuss the Householder reduction to upper Hessenberg form. The parallel algorithm requires the broadcast vector sum and the total exchange primitives.

For each algorithm, we will review the relevant theory in order to present and analyze the sequential algorithm and then discuss the development and complexity of the parallel solution. Issues pertaining to the implementation of these algorithms and the results of the experiments on the SYMULT S2010 will then be presented and compared to the theoretical expectations. Code for these routines can be found in the appendix.

In Chapter 6, we summarize the theoretical and experimental findings, discuss the potential and limitations of our approach and discuss future work and other applications.

Chapter 2

Communications

Many sequential algorithms cannot be decomposed into totally independent tasks, and distributed to separate processors for isolated computation. Instead these parallel tasks must communicate to share data and synchronize processing. It follows that improved interprocessor communications would benefit all but the most perfectly parallelizable algorithms. Typically the time to perform a communication far exceeds the time to perform a floating point operation and often the communication time of an algorithm dominates the computation time until problem sizes become very large.

In this chapter, we will examine in detail the problem of exchanging data in a ring of processors. We will describe and analyze algorithms to perform a set of data exchange operations considered sufficient to perform matrix computations. The theoretical analysis of implementing a set of primitives similar to ours on different parallel architectures is studied in [9]. By restricting communications to only these primitives the calling routines become independent of the underlying architecture. In addition, the programmer is provided with a conceptual tool with which parallel algorithms may be more easily developed.

2.1 The Primitives

In this section, we present the communication primitives some of which we will use to implement the matrix algorithms in the subsequent chapters. We will derive the time complexity of the various communication operations as they have been implemented for the ring topology. In the next sections, the term node will often be used to refer to a processor. In particular, the *source* node will refer to the processor that originates the communication and the *destination* node(s) to the processor(s) to which the message is directed.

We assume that sending a packet of m data items (e.g. double precision floating point numbers) between neighboring nodes in the ring requires time $\alpha + m\beta$, where α represents the communications startup time, and β is the per item transmission time. It is also assumed that each processor can simultaneously send to both neighbors or simultaneously send to one neighbor and receive from the other neighbor.

2.1.1 Node-to-Neighbor

Moving data from one processor to another represents the simplest data transfer operation. Here one node sends a packet of m items to a neighboring node. The time complexity for this operation is

$$\alpha + m\beta.$$

This operation is used in solving triangular systems as seen in Section 3.3 and it proves useful when performing timings and handling exceptions.

2.1.2 One-Way-Shift

This operation involves every node sending data to its right neighbor or every node sending data to its left neighbor. Each node sends a packet of m items to its neighbor. The time complexity for this operation is

$$\alpha + m\beta.$$

This operation has the same time complexity as the node-to-neighbor communication since all nodes communicate in parallel.

2.1.3 Broadcast

This data exchange operation involves transferring m data items from one processor, the source node, to all other processors. It occurs frequently in parallel numerical algorithms such as in Gaussian elimination, the QR decomposition and in the reduction of a matrix to upper Hessenberg form.

We will implement this operation in two different ways, which we will call the broadcast and the pipelined broadcast. The broadcast sends the entire message in one direction around the ring. The pipelined broadcast takes advantage of parallelism in the communication by breaking the message into ν packets and transmitting several packets simultaneously in both directions around the ring.

In the *broadcast* operation the source node initiates the communication by sending the m data items to its right neighbor. All other nodes call the broadcast receive routine which instructs them to receive data from their left neighbor and send the data on to their right neighbor (if it is not the source node). This requires $(p-1)$ sends of the m data items. The time complexity for the broadcast operation is

$$(p-1)(\alpha + m\beta).$$

This implementation may seem rather naive. However, in inherently sequential situations, where it is best for the node adjacent to the source to receive all the information first, this implementation performs very well. This situation arises in the Gaussian elimination, Cholesky and QR decomposition, and modified Gram-Schmidt algorithms presented in Chapters 3 and 4.

In the *pipelined broadcast* the source node breaks the message into ν packets and sends them in both directions around the ring. The maximum distance that a packet will travel is $\lfloor p/2 \rfloor$. Without loss of generality let the source node be $\mathbf{P}_0$. Packets sent to the right neighbor will follow the right path, $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_{\lfloor p/2 \rfloor}$, and packets sent to the left neighbor will traverse the left path, $\mathbf{P}_0, \mathbf{P}_{p-1}, \ldots, \mathbf{P}_{(\lfloor p/2 \rfloor + 1)}$. The right path is at least as long as the left path and since data transfers are occurring in parallel we need only consider the right path when considering complexity.

In step 1, $\mathbf{P}_0$ sends packet #1 to $\mathbf{P}_1$. Next, in step 2, $\mathbf{P}_0$ sends packet #2 to $\mathbf{P}_1$, and $\mathbf{P}_1$ sends packet #1 to $\mathbf{P}_2$. After $\lfloor p/2 \rfloor$ steps, packet #1 has reached $\mathbf{P}_{\lfloor p/2 \rfloor}$ and the *pipe* is filled. After $(\nu - 1)$ more steps, the last packet reaches $\mathbf{P}_{\lfloor p/2 \rfloor}$ and the broadcast is complete. The pipelined broadcast then takes $\lfloor p/2 \rfloor + \nu - 1$ steps, where each step takes time $\alpha + m\beta/\nu$. The time complexity for this operation is

$$(\lfloor p/2 \rfloor + \nu - 1)(\alpha + m\beta/\nu).$$

This time is minimized, if the optimal packet size of

$$\sqrt{\frac{m\beta(\lfloor p/2 \rfloor - 1)}{\alpha}}$$

is chosen, making the minimum time complexity,

$$\left(\sqrt{m\beta} + \sqrt{(\lfloor p/2 \rfloor - 1)\alpha}\right)^2 .$$

Note: we have not yet implemented the pipelined broadcast.

2.1.4 Total Exchange

The total exchange can be considered a multi-broadcast or $(p-1)$ consecutive one-way-shifts. Every processor sends a block of data of the same size, m, to every other processor. This communication is used in the Householder reduction to upper Hessenberg form, presented in Chapter 5.

Each node begins the communication by sending its m data items to its right neighbor. Once these messages arrive each node stores the received message then sends it on to its right neighbor. This cycle is repeated $(p-1)$ times, at which point all nodes possess all p messages. Since all nodes can send simultaneously the time complexity for this operation is

$$(p-1)(\alpha + m\beta).$$

2.1.5 Data Transpose

This operation is the simultaneous scattering of data packets from each processor to every other processor. As its name suggests, this operation acts much like the transpose operation for a matrix.

Each node $\mathbf{P}_i$ has p packets, each of size h, denoted by x_{ij}, $0 \leq j < p$. Let $m = (p-1)h$ be the total amount of data each processor must send. After the data transpose, each node $\mathbf{P}_i$ has packets x_{ji}, $0 \leq j < p$.

Every node begins the communication by sending its m data items or $(p-1)$ packets to its right neighbor. When each node receives the message of this size from its left neighbor, the node removes one packet from the message and sends the remainder, now of size $(p-2)h$ on to its right neighbor. This cycle is repeated $(p-1)$ times. During the kth cycle the message size is $m_k = (p-k)h$. Since all nodes can send simultaneously the time complexity for this operation is,

$$\sum_{k=1}^{p-1}(\alpha + (p-k)h) = (p-1)\alpha + \frac{pm}{2}\beta.$$

Note: we have not yet implemented the data transpose operation.

2.1.6 Vector Sum

In this operation each node $\mathbf{P}_i$ owns a vector of length $n = hp$, where for simplicity we now assume h is an integer. Each processor divides its vector into p equal sections, indexed by i where $0 \leq i < p$. The sum of section i from all nodes will reside on $\mathbf{P}_i$ on completion of the distributed vector sum.

All processors begin the operation by sending to their right neighbor the section of the vector whose total will eventually reside on their left neighbor. Then each processor receives the section from its left neighbor, adds to this section the corresponding section of its own vector and sends the result on to its right neighbor. This cycle is repeated $(p-1)$ times. Since all nodes can send simultaneously the time complexity for this operation is,

$$(p-1)(\alpha + h\beta + h\gamma).$$

The Householder reduction to Hessenberg form, seen in Chapter 5, requires this communication.

2.1.7 Inner Product

Two other primitives that we plan to include in this set are the inner product and global compare operations. We briefly describe them in this and the next section.

Let x and y be vectors of length $n = hp$, and assume that they are distributed among the processors so that each node has parts of x and y of length h. The inner product computes $x^T y$ leaving the result on each node.

2.1.8 Global Compare

Assume that $\xi_i \in \mathbf{P}_i$ for $i = 0, \ldots, p-1$, then the global compare operation finds the largest ξ_i and distributes it to all nodes along with the index of the processor that originally owned it.

Chapter 3

Systems of Linear Equations

In this chapter, we will consider algorithms concerned with the problem of solving dense linear systems of equations, $Ax = b$, where $A \in \mathbf{R}^{n \times n}$ and $x, b \in \mathbf{R}^n$. Sequential algorithms which solve this problem are presented in [2, 11] and efficient implementations of these algorithms can be found in [1].

In Sections 3.1 and 3.2 we examine algorithms to factor a matrix into the product of a lower and upper triangular matrix or LU decomposition. Such decompositions are based on the existence of a unique LDU decomposition of a square general matrix A. In a LDU decomposition, L is unit lower triangular, D is diagonal and U is unit upper triangular. The LU decompositions differ in how the diagonal matrix is handled in the LDU decomposition. For dense matrices, $O(n^3)$ operations are required to perform these decompositions.

In Section 3.3 we consider the solution of the resulting triangular systems. Once the LU-decomposition has been obtained the problem, $LUx = b$, reduces to solving two triangular systems,

$$Ly = b$$

and

$$Ux = y.$$

The first system is solved by forward elimination and then the second can be solved by backward substitution. Each of these algorithms requires $O(n^2)$ operations and contribute little to the overall complexity compared to the decomposition. However, in practice it is common for the system $Ax = b_i$ to be solved for many different right hand side vectors, b_i. In this case, the matrix A is decomposed once and the repeated triangular solves become a significant part of the complexity. We implement and test a parallel back-substitution algorithm to solve an upper triangular system. The algorithm for a lower triangular system is analogous.

3.1 Gaussian Elimination with Partial Pivoting

The first of these algorithms is Gaussian elimination with partial pivoting for factoring a permuted general matrix PA into the product of a unit lower triangular matrix and an upper triangular matrix. In terms of the LDU decomposition, $PA = LDU = LU'$ where $U' = DU$ is upper triangular.

DEFINITION 3.1 *An* elementary lower triangular *matrix of order n and index k is of the form,*

$$M = I_n - me_k^T$$

where $m = (0, \dots, 0, \mu_{k+1}, \mu_{k+2}, \dots, \mu_n)^T$.

Elementary transformations are elementary lower triangular matrices that are used to introduce zero components in a vector and can be exploited to perform the reduction by Gaussian elimination. The following theorem states this property more precisely.

THEOREM 3.1 *Given a vector $x = (\xi_1, \dots, \xi_k, \dots, \xi_n)^T$ and $\xi_k \neq 0$, there exists an elementary transformation, M such that $Mx = (\xi_1, \dots, \xi_k, 0, \dots, 0)^T$. If $\xi_k = 0$, then M does not exist unless $\xi_{k+1}, \dots, \xi_n$ are also zero.*

The following theorem states the existence of this factorization.

THEOREM 3.2 *Let A be an $n \times n$ matrix. Then there are elementary permutations P_i $(i = 1, 2, \dots, n-1)$, and elementary lower triangular matrices M_i of index i $(i = 1, 2, \dots, n-1)$, such that*

$$A_n = M_{n-1}M_{n-2}\dots M_1 P_{n-1}P_{n-2}\dots P_1 A$$

is upper triangular.

3.1.1 Sequential Algorithm

In Gaussian elimination, a matrix A is reduced to upper triangular form by premultiplying A by a sequence of elementary transformations, $M_{n-1}, \dots, M_1$, so as to introduce zeros below the diagonal of the product matrix. The process begins by producing the elementary transformation M_1 and premultiplying $A^{(1)} = A$ to get $A^{(2)} = M_1 A^{(1)}$. The matrix $A^{(2)}$ has zeros below the diagonal in its first column. In the kth step, $(k = 1, \ldots, n-1)$, of the algorithm the matrix $A^{(k)}$ has zeros below the diagonal in the first $(k-1)$ columns. The matrix, $M_k = I_n - m_k e_k^T$ is determined by the vector,

$$m_k = (0, \ldots, 0, \mu_{k+1,k}, \ldots, \mu_{nk}),$$

where

$$\mu_{ik} = \alpha_{ik}^{(k)} / \alpha_{kk}^{(k)}, \qquad (i = k+1, \ldots, n).$$

Here the elements μ_{ik} are called the *multipliers* and α_{kk}, the kth *pivot* element.

Notice that because of the structure of M_k, premultiplying $A^{(k)}$ by M_k does not change the first $(k-1)$ columns or rows of $A^{(k)}$. In fact $A^{(k+1)}$ is identical to $A^{(k)}$ in the first k rows and $k-1$ columns. The effect of the kth step of the algorithm is to annihilate the subdiagonal elements in the kth column and to perform a rank one update on $A_{k+1,k+1}^{(k)}$, the $(n-k) \times (n-k)$ trailing principal submatrix of $A^{(k)}$. In the implementation the subdiagonal zero elements can be overwritten by the multipliers which were used to create the zeros. Premultiplication by M_k then alters the matrix $A^{(k)}$ to produce the matrix $A^{(k+1)}$ as follows,

$$A^{(k+1)} = M_k A^{(k)} = \begin{pmatrix} I_{k-1} & 0 \\ 0 & M_k' \end{pmatrix} \begin{pmatrix} A_{11}^{(k)} & A_{12}^{(k)} \\ 0 & A_{kk}^{(k)} \end{pmatrix}.$$

Here M_k' is an elementary transformation of index 1, and

$$A_{kk}^{(k)} = \begin{pmatrix} \alpha_{kk}^{(k)} & \left(a_{k,k+1}^{(k)}\right)^T \\ a_{k+1,k}^{(k)} & A_{k+1,k+1}^{(k)} \end{pmatrix}.$$

This results in $A^{(k+1)}$ having the following form,

$$A^{(k+1)} = \begin{pmatrix} A_{11}^{(k)} & A_{12}^{(k)} \\ 0 & \begin{pmatrix} \alpha_{kk}^{(k)} & (a_{k,k+1}^{(k)})^T \\ 0 & A_{k+1,k+1}^{(k+1)} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} \\ 0 & A_{k+1,k+1}^{(k+1)} \end{pmatrix},$$

where the multipliers are given by

$$\mu_{k+1,k}^{(k+1)} = a_{k+1,k}^{(k)} / \alpha_{kk}^{(k)}$$

and

$$A_{k+1,k+1}^{(k+1)} = A_{k+1,k+1}^{(k)} - \mu_{k+1,k}^{(k+1)} (a_{k,k+1}^{(k)})^T.$$

If the kth pivot element is zero, the process cannot proceed unless all the sub-diagonal elements of the kth column of $A^{(k)}$ are also zero. Furthermore, the

Algorithm 3.1 *The following algorithm uses Gaussian elimination with partial pivoting to overwrite A with the LU decomposition of PA. The pivot row indices are stored in a separate vector of length $(n-1)$.*

```
$A_1 = A$
for $k = 1, \ldots, n-1$
    find pivot to determine $P_k$
    compute $M_k = I_n - m_k e_k^T$
    $A \leftarrow A^{(k+1)} = M_k P_k A^{(k)}$
```

Figure 3.1: Gaussian Elimination with Partial Pivoting

algorithm becomes unstable if the pivot element is close to zero, since this can make the multipliers large and cause subtractive cancellation when updating elements in the submatrix [2, 11]. However, the algorithm can be made more stable by *partial pivoting* or swapping rows of the submatrix so that the largest element on or below the diagonal in the kth column is moved to the diagonal position. Performing a pivot with the ith row during the kth step of the algorithm ($i \geq k$) is equivalent to premultiplying the matrix $A^{(k)}$ by a permutation matrix P_k, which is of the form

$$P_k = \begin{pmatrix} I_{k-1} & 0 \\ 0 & P_{i1} \end{pmatrix},$$

where P_{i1} differs from I_{n-k+1} only in that the first and ith rows are interchanged. With pivoting, the kth step amounts to premultiplying the matrix $A^{(k)}$ by the permutation matrix P_k and the elementary triangular matrix M_k so that,

$$A^{(k+1)} = M_k P_k A^{(k)}$$

has zeros below the diagonal in the first k columns.

The result of this process is $M_{n-1}P_{n-1}\ldots M_1P_1A = U$, where U is an upper triangular matrix. Letting $P = P_{n-1}\ldots P_1$ and noting that $P_i^{-1} = P_i$, we have, $PA = P(M_{n-1}P_{n-1}\ldots M_1P_1)^{-1}U = M_1^{-1}\ldots M_{n-1}^{-1}U$ or $PA = LU$, where $L = M_1^{-1}\ldots M_{n-1}^{-1}$ is a unit lower triangular matrix. The elements of L below the diagonal are the multipliers μ_{ij} produced when determining M_j to zero out the jth column of $A^{(j)}$.

The time to find the pivot and compute the multipliers is insignificant for large n, compared to the last step of the loop (see Figure 3.1), in which the submatrix is updated, requiring $(n-k)^2$ flops. If we sum this expression over all iterations we get,

$$\mathrm{T}_1(n) \approx \sum_{k=1}^{n-1}(n-k)^2\gamma \approx \frac{n^3}{3}\gamma. \qquad (3.1)$$

Ignoring low order terms, this algorithm requires approximately $n^3/3$ flops.

3.1.2 Parallel Algorithm

We begin our discussion of the parallel algorithm to perform Gaussian elimination with pivoting by considering the distribution of the problem in a *column wrapped* fashion among a ring of p processors, $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$. As described in Section 1.2, the first column is assigned to $\mathbf{P}_0$, the pth column to $\mathbf{P}_{p-1}$, the $(p+1)$th column to $\mathbf{P}_0$ and so on. If the number of processors divides the number of columns, n, then each processor receives an equal share of the matrix. Otherwise $(n \bmod p)$ processors will have $\lceil n/p \rceil$ columns and the rest will have $\lfloor n/p \rfloor$ columns.

Given that each processor has approximately (n/p) columns, we consider the sequential algorithm, which loops through a set of computations for each of the first $(n-1)$ columns of the matrix. Within each iteration, the sequence of computations is: find the pivot row, determine the multipliers, and update the submatrix. The pivot row and multipliers can all be determined by processor $\mathbf{P}(k)$, the one owning the kth column. The submatrix, however, is distributed among all processors, so all processors are required to assist in applying the update.

First, note the need for synchronization and communication. There is a required order to the computations that is satisfied by completely applying the kth update to column j before applying the $(k+1)$th update to that same column. The kth iteration begins with finding the kth pivot row and then computing the multipliers before the $(n-k) \times (n-k)$ submatrix is updated. The columns of the submatrix can be updated in any order and hence the update can be performed in parallel by all processors once they have received the multipliers and pivot row index from $\mathbf{P}(k)$. This requires a communication from one processor to all others, each iteration. The *broadcast* primative can be used to disperse this information as well as to synchronize the computations among all processors so that the order of computations is preserved.

Algorithm 3.2 *This algorithm uses the broadcast primitive alone to reduce a general matrix $A \in \mathbf{R}^{n \times n}$ to triangular form using elementary transformations with partial pivoting overwriting PA with LU. Pseudo code driving node $\mathbf{P}_i$ in a ring of processors is given by,*

```
for $k = 1, \ldots, n-1$
    if $k \in \mathbf{P}_i$
        find pivot
        compute $M_k = I_n - m_k e_k^T$                    $[(n-k)\gamma]$
        broadcast $m_k$ and pivot index                    $[2(\alpha + (n-k+1)\beta)]$
    else
        receive $m_k$ and pivot index
    update $a_j \leftarrow a_j - \alpha_{kj} m_k,\ j \in \mathbf{P}_i > k$    $[(\lceil \frac{n-k}{p} \rceil (n-k))\gamma]$
```

Figure 3.2: Parallel Gaussian Elimination with Partial Pivoting

One more observation before we specify the algorithm for any processor. We recall from Section 2.1.3 that the non-pipelined broadcast primative involved the source node sending the complete message to its right neighbor in the ring. This right neighbor then receives the message and passes it on to its own right neighbor and so on, until the message reaches the left neighbor of the source node. Returning to the parallel algorithm for Gaussian elimination, during the kth iteration processor $\mathbf{P}(\mathrm{k})$ computes the pivot row and multipliers, then broadcasts this data to the other processors. $\mathbf{P}(k+1)$ is the first processor to receive the data and will continue the broadcast operation by sending the data on to $\mathbf{P}(k+2)$ at which point $\mathbf{P}(k+1)$ is free to begin updating its part of the submatrix. The effect of implementing the broadcast in this manner is an overlap of the remainder of the broadcast with computation. As soon as $\mathbf{P}(k+1)$ completes updating its part of the submatrix it may determine the $(k+1)$th pivot row, compute the multipliers, and begin the next broadcast. As the algorithm nears completion, (i.e, $k > (n-p)$) the data need not be broadcast to all other processors since the submatrix is contained in fewer than p processors' memories. However, this contributes little to the time complexity and can be overlooked in favor of the simplicity of the algorithm.

In Figure 3.2 the expressions in brackets indicate the effective contribution to the time complexity by that step in each iteration. Notice that, as discussed above, the effective time to perform the broadcast is equivalent to the time to perform two node-to-neighbor communications with a $(n-k+1)$ size vector. Here we assume that the sending processor is not free to compute until the message is received. The time complexity for a single iteration is then,

$$(n-k)\gamma + 2(\alpha + (n-k+1)\beta) + \left(\left\lceil \frac{n-k}{p} \right\rceil (n-k)\gamma\right).$$

Noting that $\lceil x \rceil \leq (x+1)$, and summing over the $(n-1)$ iterations of the loop we arrive at an upper bound for the total time complexity.

$$\mathrm{T}_p(n) \leq \frac{\gamma}{p}\sum_{k=1}^{n-1}(n-k)^2 + 2\gamma\sum_{k=1}^{n-1}(n-k) + 2(n-1)\alpha + 2\beta\sum_{k=1}^{n-1}(n-k+1)$$

Performing this sum and ignoring low order terms we get

$$\mathrm{T}_p(n) \approx \left(\frac{n^3}{3p} + n^2\right)\gamma + 2n\alpha + n^2\beta. \tag{3.2}$$

This compares favorably with the sequential time complexity, given by (3.1). For a fixed value of p, as the problem size increases we have

$$\lim_{n\to\infty}\frac{\mathrm{T}_1(n)}{\mathrm{T}_p(n)} = p,$$

so the speedup approaches p and efficiency nears 100%. In (3.2) we retain the term, $n^2\gamma$, because of its increasing significance as p approaches n. This term results from the cost of computing M_k sequentially each iteration and as p approaches n it contributes, along with communication cost, to the inefficiency.

3.1.3 Implementation and Numerical Experiments

The routine PGEFA (parallel general matrix factorization) drives each node in a ring of processors to factor a square general matrix by Gaussian elimination. The calling sequence and argument descriptions are given in Figure 3.3.

Figure 3.4 displays the expected and observed timings that were obtained by PGEFA with a ring of eight processors on the Symult S2010 for $n = 10, 20, \ldots, 490$. Figure 3.5 plots the ratio of these observed results over the expected times. Discrepancies between these times are explained by the combined effects of several factors. For smaller problems, the effect of having dropped negative low order terms when approximating the time complexity tends to make the expected time greater than the

PGEFA(N,A,IPVT,INFO)

- On Entry

  N is the order of the matrix to be factored.

  A is a pointer to a vector of type FLT which contains columns of the matrix to be factored.

- On Return

  A is a pointer to columns of the LU decomposition where U is upper triangular and L is unit lower triangular whose elements are the multipliers used to obtain U.

  IPVT is a pointer to a vector of integers which are the indices of the pivot rows.

  INFO is a pointer to an integer, which has value 0 if the matrix A is non-singular or has value k if the kth pivot element is zero.

Figure 3.3: PGEFA

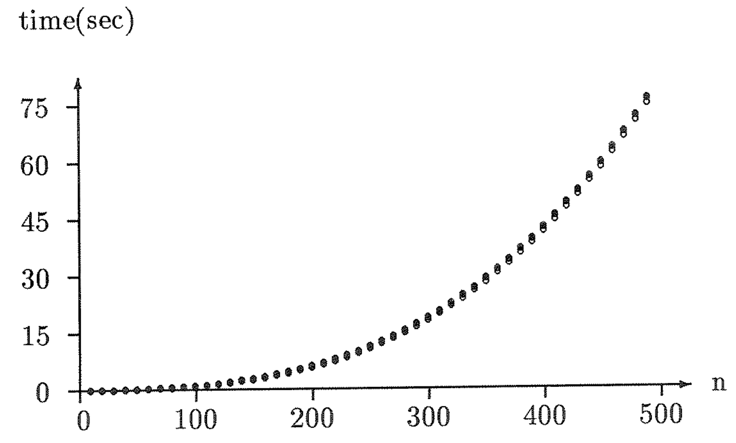

Figure 3.4: Observed and expected timings for $p = 8$ during Gaussian Elimination with Partial Pivoting (key:: •: observed, o: expected)

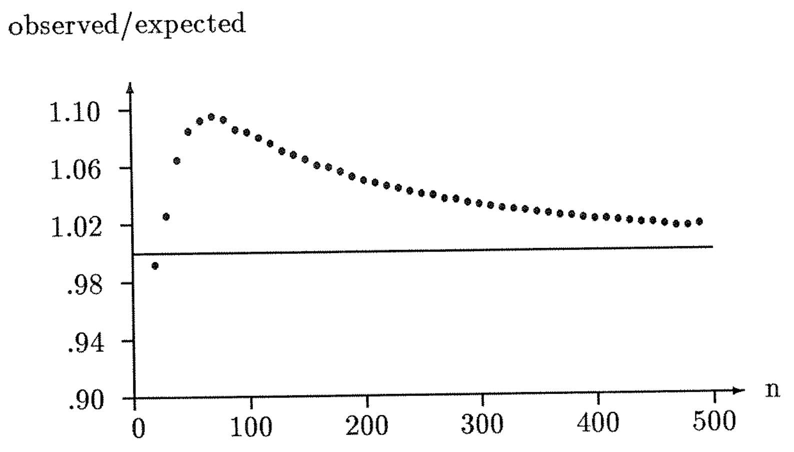

Figure 3.5: Ratio of Observed to Expected run times for Gaussian elimination with 8 processors

observed time. Also, for small n, γ is underestimated owing to the overhead in calling the BLAS routines. This tends to make the expected time less than the observed time. As the problem size increases this underestimation becomes less pronounced. In addition, the value used for α is an overestimation when n is small, because of a message buffer size of 256 bytes [15]. This tends to make the expected time greater than the observed time for message sizes less than 30 double precision floating point numbers. The improvement after $n > 70$ is caused by the dominating influence of the $(n^3/3)\gamma/p$ term, which eventually overshadows all other terms contributing to the time complexity. This agreement is better for fewer than eight processors and slightly worse for 16 or 24 processors.

Figure 3.6 plots the efficiencies attained by PGEFA for various ring and problem sizes. Reasonable efficiencies ($> 50\%$) are obtained once $n/p > 10$.

3.2 Cholesky Decomposition

We will now consider the decomposition of a symmetric positive definite matrix A into the product of a lower triangular matrix and its transpose. In terms of the LDU decomposition, $A = LDU = LDL^T = LD^{1/2}D^{1/2}L^T$. Letting $L' = LD^{1/2}$, we have $A = L'(L')^T$ where L' is a lower triangular matrix. This decomposition is

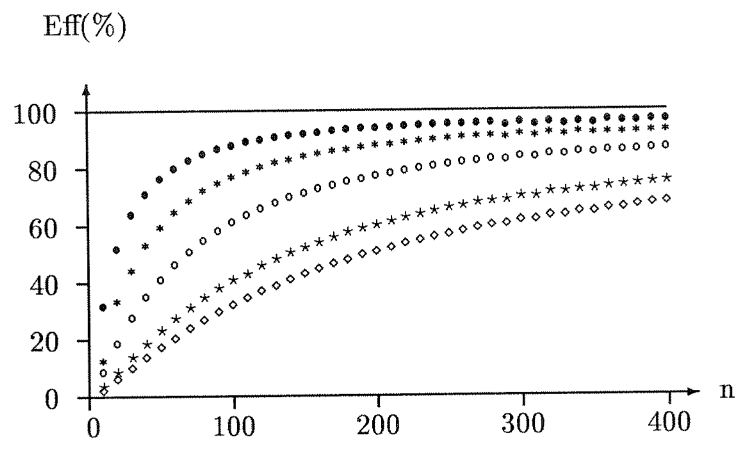

Figure 3.6: Observed efficiencies attained during Gaussian Elimination with Partial Pivoting (key:: $\bullet$: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

known as the Cholesky decomposition and L' as the Cholesky triangle.

Obviously, this decomposition does not exist for every matrix; A must be symmetric and the elements of D must be nonnegative limiting A to the class of matrices which are symmetric positive semi-definite. The algorithms presented in Figures 3.7 and 3.8 require that A be positive definite.

DEFINITION 3.2 *A symmetric matrix, $A \in \mathbf{R}^{n\times n}$, is* positive definite *if and only if for every nonzero $x \in \mathbf{R}^n$,*

$$x^T Ax > 0.$$

THEOREM 3.3 *Given that $A \in \mathbf{R}^{n\times n}$ is symmetric and positive definite, there is a unique lower triangular matrix L with positive diagonal elements such that $A = LL^T$.*

3.2.1 Sequential Algorithm

The proof of the existence of the Cholesky decomposition for positive definite matrices is constructive in that it suggests an algorithm with which we can compute the Cholesky triangle. However, we can also arrive at an algorithm to

Algorithm 3.3 *The following algorithm computes the Cholesky triangle by column and overwrites the lower half of A with L.*

```
for $k = 1, \ldots, n$
    $\lambda_{kk} = \alpha_{kk} \leftarrow (\alpha_{kk})^{1/2}$
    $\lambda_{ik} = \alpha_{ik} \leftarrow \alpha_{ik}/\lambda_{kk}$                    $(i > k)$
    for $j = k+1, \ldots, n$
        for $i = j, \ldots, n$
            $\alpha_{ij} \leftarrow (\alpha_{ij} - \lambda_{ik}\lambda_{jk})$
```

Figure 3.7: Cholesky Decomposition

compute L by considering entries of $A = LL^T$,

$$\alpha_{ij} = \sum_{k=1}^{\min(i,j)} \lambda_{ik}\lambda_{jk},$$

since the (k,j)th element of L^T is λ_{jk}. If we consider only elements in the lower triangle, where $i \geq j$ we have,

$$\alpha_{ij} = \sum_{k=1}^{j} \lambda_{ik}\lambda_{jk} = \left(\sum_{k=1}^{j-1} \lambda_{ik}\lambda_{jk}\right) + \lambda_{ij}\lambda_{jj}$$

which after rearranging terms yields the following equations,

$$\lambda_{ij} = \left(\alpha_{ij} - \sum_{k=1}^{j-1} \lambda_{ik}\lambda_{jk}\right) / \lambda_{jj} \qquad (i > j)$$

and

$$\lambda_{jj} = \left(\alpha_{jj} - \sum_{k=1}^{j-1} \lambda_{jk}^2\right)^{1/2}.$$

We now vary the order in which these computations are performed so that as the kth column of L is formed, the associated update of the lower half of the $(n-k) \times (n-k)$ trailing submatrix of A is applied before the $(k+1)$th column is formed [8], much like Gaussian elimination. The result is given by Algorithm 3.7.

Most of the work is done in the loop indexed by j, updating the lower *half* of the $(n-k) \times (n-k)$ submatrix. There are $((n-k)^2 + (n-k))/2$ elements in this portion of the submatrix. Summing this over all iterations gives,

$$T_1(n) \approx \frac{1}{2}\sum_{k=1}^{n}((n-k)^2 + (n-k))\gamma \approx \frac{n^3}{6}. \tag{3.3}$$

Ignoring the lower order terms, this algorithm requires approximately $n^3/6$ flops for large n. By taking advantage of symmetry, we are able to approximately halve the number of computations required by Gaussian elimination to factor a matrix into a product of triangular matrices.

3.2.2 Parallel Algorithm

Again we distribute the matrix A to the ring of processors in a column wrapped fashion. The parallelization of the algorithm proceeds almost identically to that for Gaussian elimination. Given that each processor has approximately (n/p) columns, we consider the sequential algorithm which loops through a set of computations to form each of the n columns of the matrix L. Within each iteration, the sequence of computations is to first determine λ_{kk}, then scale the kth column, and then update the lower half of the submatrix. In the kth iteration, the kth column of L can be completely determined by $\mathbf{P}(k)$, the processor that owns the kth column of A. The submatrix however, is distributed among all processors, so all processors are required to assist in applying the update. The (i,j)th element is updated as follows,

$$\alpha_{ij} \leftarrow (\alpha_{ij} - \lambda_{ik}\lambda_{jk}).$$

For a processor to update the jth column of the submatrix it must have the newly formed kth column of L. This requires a broadcast by $\mathbf{P}(k)$ of the kth column to all other processors. The updating of the submatrix can be performed in parallel by all processors owning columns of that submatrix once they have received the vector $(\lambda_{kk}, \ldots, \lambda_{nk})^T$ from $\mathbf{P}(k)$.

Again, in Figure 3.8, the expressions in brackets indicate the effective contribution to the time complexity of that step in each iteration. The scaling of the kth column takes $(n-k)\gamma$ time and, as was the case in Gaussian elimination, the effective time to perform the broadcast is equivalent to the time to perform two node-to-neighbor communications with a $(n-k+1)$ size vector. Continuing with this reasoning, the $(k+1)$th iteration may begin as soon as $\mathbf{P}(k+1)$ completes

Algorithm 3.4 *This algorithm drives processor* $\mathbf{P}_i$ *in a ring of processors to factor a symmetric positive definite matrix by the Cholesky algorithm. The matrix is distributed among the processors in a column wrapped fashion.*

```
for $k = 1, \ldots, n$
    if $k \in \mathbf{P}_i$
        $\lambda_{kk} = \alpha_{kk} \leftarrow (\alpha_{kk})^{1/2}$
        $\lambda_{ik} = \alpha_{ik} \leftarrow \alpha_{ik}/\lambda_{kk}$        $(i > k)$    $[(n-k)\gamma]$
        broadcast $(\lambda_{kk}, \ldots, \lambda_{nk})^T$                $[2(\alpha + (n-k+1)\beta)]$
    else
        receive $(\lambda_{kk}, \ldots, \lambda_{nk})^T$
    for $j \in \mathbf{P}_i$ and $j > k$
        $\alpha_{ij} \leftarrow (\alpha_{ij} - \lambda_{ik}\lambda_{jk})$        $(i \geq j)$    $[\frac{1}{2p}(n-k)(n-k+p)\gamma]$
```

Figure 3.8: Parallel Cholesky Decomposition

updating the lower half of columns that it owns and that belong to the submatrix (i.e.,columns $j \ni j \in \mathbf{P}(k+1) \wedge j > k$). Processor $\mathbf{P}(k+1)$ owns $\lceil (n-k)/p \rceil$ columns of this submatrix, and these columns have indices $j = k + 1 + ip$, for $i = 0, 1, \ldots, \lceil (n-k)/p \rceil - 1$. In column j, there are $(n - j + 1)$ elements to update. The time for $\mathbf{P}(k+1)$ to update its share of the submatrix is then,

$$\sum_{i=0}^{\lceil \frac{n-k}{p} \rceil - 1} (n - k - ip)\gamma,$$

and using the approximation $\lceil (n-k)/p \rceil \approx (n-k)/p + 1$ yields the third expression in brackets,

$$\sum_{i=0}^{\frac{n-k}{p}} (n - k - ip)\gamma = \frac{1}{2p}(n-k)(n-k+p)\gamma.$$

Summing these three expressions over all iterations gives the total time complexity of the algorithm,

$$\begin{aligned} \mathrm{T}_p(n) &\approx \sum_{k=1}^{n} \Big[\frac{1}{2p}(n-k)(n-k+p)\gamma + (n-k)\gamma \\ &\qquad + 2(\alpha + (n-k+1)\beta \Big] \\ &\approx \left(\frac{n^3}{6p} + \frac{3n^2}{4} \right) \gamma + 2n\alpha + n^2\beta. \end{aligned} \tag{3.4}$$

PPOFA(N,A,INFO)

- On Entry

  N is the order of the matrix to be factored.

  A is a pointer to a vector of type FLT which contains columns of the matrix to be factored.

- On Return

  A is a pointer to columns of L, the Cholesky triangle, in the lower triangle of A; the upper triangle remains unchanged.

  INFO is a pointer to an integer, which has value 0 if the factorization completed normally or has value k if the leading principal submatrix of order k is found not to be positive definite.

Figure 3.9: PPOFA

This algorithm also approaches 100% efficiency for large n and fixed p. Comparing equations (3.3) and (3.4) we see that,

$$\lim_{n\to\infty} \frac{\mathrm{T}_1(n)}{\mathrm{T}_p(n)} = p.$$

However, as p approaches n the two terms $n^2\beta$ and $(3n^2/4)\gamma$ gain significance and efficiency drops. The term $(3n^2/4)\gamma$ reflects the cost of the sequential portion of each iteration, where one processor determines λ_{kk} and scales the kth column.

3.2.3 Implementation and Numerical Experiments

The routine PPOFA (**par**allel **po**sitive definite matrix **fa**ctorization) drives each node in a ring of processors to factor a symmetric positive definite matrix by the Cholesky algorithm. The calling sequence and argument descriptions are given in Figure 3.9.

Figure 3.10 plots the observed timings obtained by PPOFA with various ring sizes on the Symult S2010 for problems ranging up to $n = 490$. Figure 3.11 plots the expected and observed timings obtained by PPOFA with a ring of eight processors on the Symult S2010 for problem sizes up to $n = 490$. The expected and observed timings agree within 12%. Again, discrepancies between these times are

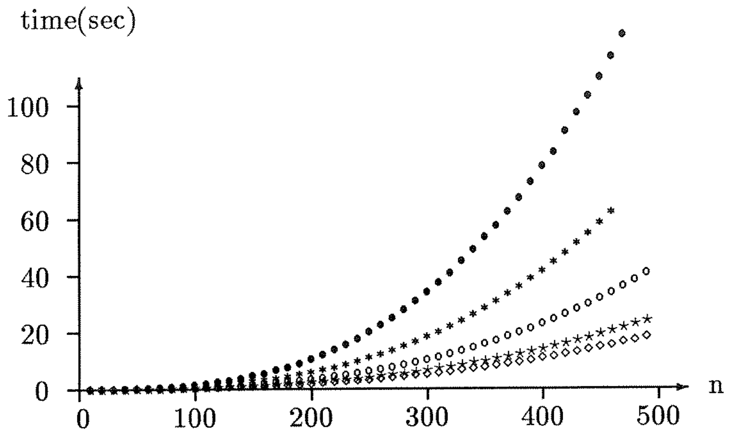

Figure 3.10: Observed timings attained during Cholesky Decomposition (key:: •: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

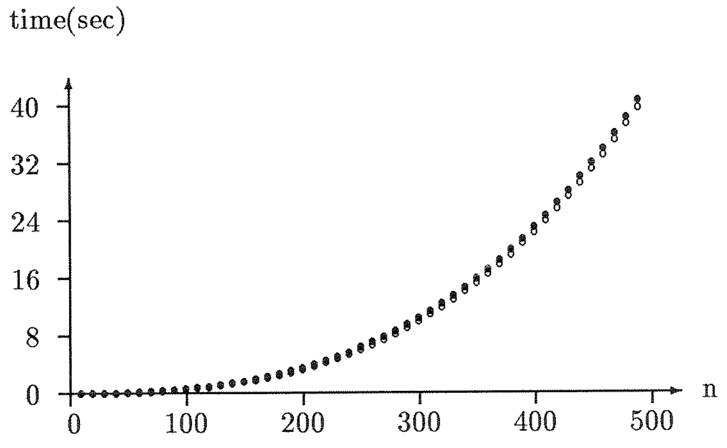

Figure 3.11: Observed and expected timings for $p = 8$ during Cholesky Decomposition (key:: •: observed, $\circ$: expected)

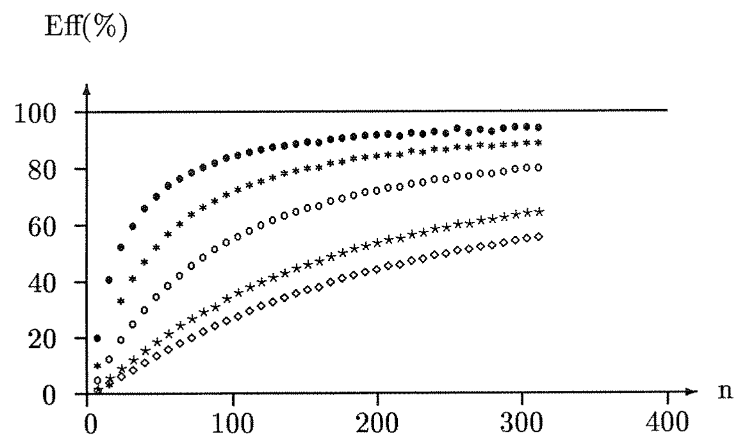

Figure 3.12: Observed efficiencies attained during Cholesky Decomposition (key:: •: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

explained by the combined effects of several factors. This agreement is better for fewer than eight processors and slightly worse for 16 and 24 processors. Within the range of tests performed, it is as low as .1% when $p = 2$ and $n = 400$, and as high as 20% when $p = 24$ and $n = 130$.

Figure 3.12 plots the observed efficiency attained by PPOFA for various ring sizes and problem sizes. Reasonable efficiencies can be obtained once $n/p > 15$. The Cholesky algorithm is initially less efficient than that for Gaussian elimination. The submatrix update is the portion of each iteration that is performed in parallel. Since in Cholesky's algorithm there is less to update, the ratio of useful computation to communication is smaller and efficiency suffers.

3.3 Triangular Solve

In this section we will discuss only the algorithm for backward substitution, which solves the following system for x,

$$Ux = b,$$

where $U \in \mathbf{R}^{n\times n}$ is upper triangular and $x, b \in \mathbf{R}^n$. The algorithm for a lower triangular system is analogous.

3.3.1 Sequential Algorithm

The algorithm to solve an upper triangular system of linear equations is an $O(n^2)$ operation. It is best introduced by first considering a 2×2 system. In this case we have,

$$\begin{pmatrix} v_{11} & v_{12} \\ 0 & v_{22} \end{pmatrix} \begin{pmatrix} \xi_1 \\ \xi_2 \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}.$$

The unknowns can be found provided $v_{11}v_{22} \neq 0$, by first solving the equation,

$$\xi_2 = \beta_2 / v_{22}$$

for ξ_2 and then substituting this value into

$$v_{11}\xi_1 + v_{12}\xi_2 = \beta_1,$$

and reducing this to a single equation with a single unknown,

$$\xi_1 = (\beta_1 - v_{12}\xi_2)/v_{11}.$$

Larger systems of order n can be solved in a similar manner, by solving for the last element of x and then updating the first $(n-1)$ elements of b, thereby reducing the remaining system to an upper triangular system of size $(n-1)\times(n-1)$. The process is repeated n times; each time a new element of x is found, the problem size is reduced to a smaller triangular system. The jth step of the algorithm is outlined below for $j = n, \ldots, 1$ and the algorithm is listed in Figure 3.3.1. We enter the jth iteration with the $j \times j$ system $Ux = b$, which can be partitioned as

$$\begin{pmatrix} U_{11} & \tilde{u}_j \\ 0 & v_{jj} \end{pmatrix} \begin{pmatrix} \tilde{x} \\ \xi_j \end{pmatrix} = \begin{pmatrix} \tilde{b} \\ \beta_j \end{pmatrix}$$

where,

$$\tilde{b}_j = (v_{1j}, \ldots v_{j-1,j})^T, \quad \tilde{x} = (\xi_1, \ldots \xi_{j-1})^T \text{ and } \tilde{b} = (\beta_1, \ldots \beta_{j-1})^T.$$

If $v_{jj} \neq 0$ we can solve for

$$\xi_j = \beta_j / v_{jj}$$

and then reduce the problem to a $(j-1)\times(j-1)$ upper triangular system by updating the first $(j-1)$ elements of b with

$$\tilde{b} \leftarrow \tilde{b} - \xi_j \tilde{u}_j$$

Algorithm 3.5 *The following algorithm solves the $n \times n$ upper triangular system of linear equations, $Ux = b$, provided $U = (u_1, \dots u_n)$ is non-singular.*

```
for $j = n, \dots, 1$
    $\xi_j \leftarrow \beta_j / v_{jj}$
    $\tilde{b} \leftarrow \tilde{b} - \xi_j \tilde{u}_j$                $\tilde{b}, \tilde{u}_j \in \mathbf{R}^{j-1}$
```

Figure 3.13: Back-Substitution

Algorithm 3.6 *This algorithm drives processor $\mathbf{P}_i$ in a ring of processors to solve the $n \times n$ upper triangular system of linear equations, $Ux = b$, provided $U = (u_1, \dots u_n)$ is non-singular. The matrix is distributed among the processors in a column wrapped fashion.*

```
for $j = n, \dots, 1$
    if $j \in \mathbf{P}_i$
        if $(j \neq n)$ receive($b$)
        $\xi_j \leftarrow \beta_j / v_{jj}$
        $b \leftarrow b - \xi_j u_j$
        if $(j \neq 1)$ send left($b$)
```

Figure 3.14: Distributed Back-Substitution

and then letting $U \leftarrow U_{11}$, $x \leftarrow \tilde{x}$, and $b \leftarrow \tilde{b}$. Clearly, the solution exists and is unique if and only if the diagonal elements of U are non-zero.

In each pass through the loop, there is a divide operation and a vector update (saxpy) operation of order $(j-1)$, totaling to approximately j flops. If we sum these over all n iterations we get a time complexity of

$$\mathrm{T}_1(n) \approx \frac{n^2}{2}\gamma. \tag{3.5}$$

3.3.2 Parallel Algorithm

The matrix U is distributed to the ring of processors in a column wrapped fashion and the right hand side vector, b, is given to $\mathbf{P}(n)$, the processor that holds the nth column of U.

First, we consider a distributed version of the sequential algorithm to motivate the parallel algorithm (see Figure 3.14). In this algorithm we let $U = (u_1, u_2, \dots, u_n)$

be a column partitioning of U. The vector b starts at $\mathbf{P}(n)$ and is passed around the ring from $\mathbf{P}(n)$ to $\mathbf{P}(n-1)$to... to $\mathbf{P}(1)$. At processor $\mathbf{P}(j)$, b is used to determine the jth component of x and is then updated before being passed on to the next processor. In this distributed algorithm, only one processor computes at any given time, hence there is no parallel activity. However, a slight modification allows processors to compute simultaneously. The resulting parallel algorithm was developed, analyzed and tested by Li and Coleman [5, 6].

In each loop iteration of the distributed sequential algorithm, the majority of work is done in the update of the vector b. The object then is to have all processors performing these updates simultaneously. However, each update of b requires that a new element of x be determined and these elements must be determined in order. For processor $\mathbf{P}(j)$ to compute ξ_j, it must have access to β_j, $\xi_{j+1}, \ldots \xi_n$, and $v_{j,j+1}, \ldots v_{jn}$. These data, however, are distributed among several processors.

Access to these values is provided by the auxiliary vectors s (sum) and t (partial sum). Initially, on processor $\mathbf{P}(n)$, s contains the last $(p-1)$ elements of b,

$$\sigma_k = \beta_{n-k+1} \qquad\qquad \texttt{for } k = 1, \ldots, p-1$$

and t contains the first $(n-p+1)$ elements of b,

$$\tau_k = \beta_k \qquad\qquad \texttt{for } k = 1, \ldots, n-p+1.$$

These vectors are initialized to zero on all other processors. As the algorithm proceeds, for any processor, τ_k will contain a partial sum of those terms to be divided by v_{kk} to determine ξ_k, that are resident on that particular processor, (including β_k on $\mathbf{P}(n)$). The vector s is used as a communications buffer in which the partial sums of all other contributing processors are gathered for determining the next $(p-1)$ elements of the vector x.

Processor $\mathbf{P}(j)$ loops through the following sequence of steps:

- receive the vector s from $\mathbf{P}(j+1)$, $(j \neq n)$, which contains partial sums used to determine $\xi_j, \ldots, \xi_{j+p-2}$,
- determine ξ_j,
- shift s so that the next element of t is entered into the last position of s and add to each element of s both partial sums from t and the new term containing ξ_j,

Algorithm 3.7 *This algorithm drives processor $\mathbf{P}_i$ in a ring of processors to solve the upper triangular system of equations $Ax = b$ by back substitution. The matrix is distributed among the processors in a column wrapped fashion. On completion the vector x is distributed among the processors such that $\xi_j \in \mathbf{P}(j)$.*

```
for $j = n, \ldots, 1$
    if $j \in \mathbf{P}_i$
        if $(j \neq n)$ receive($s$)
        $\xi_j \leftarrow (\sigma_1 + \tau_j)/v_{jj}$
        for $k = 1, \ldots, p-2$
            $\sigma_k = \sigma_{k+1} - v_{j-k-1,j}\xi_j + \tau_{j-k-1}$
        $\sigma_{p-1} = -v_{j-p+1,j}\xi_j + \tau_{j-p+1}$
        if $(j \neq 1)$ send left($s$)
        for $k = 1, \ldots, j-p+1$
            $\tau_k = \tau_k - v_{kj}\xi_j$
```

Figure 3.15: Parallel Back-Substitution

- send s to $\mathbf{P}(j-1)$, $(j \neq 1)$,
- use ξ_j to update the remaining elements of t.

This algorithm is presented in Figure 3.15 in greater detail.

Consider the following example where $p = 4$ and $n = 16$. Figure 3.16 shows the assignment of processors to columns as well as the data flow between processors via the vector s. Each column represents the elements of the vectors t and s on the processor denoted above the column during the iteration indicated below the column. In the jth column $(j = 16, \ldots, 1)$, the "x" identifies the element of s used to determine ξ_j. The three elements labeled "s" identify the elements of the buffer s which are updated and sent to $\mathbf{P}(j-1)$. The position of each letter in the triangle corresponds to the position of the element of U which is used to update that particular element of s, t or x in that iteration.

Let us examine how the data becomes available to $\mathbf{P}_1$ so that it can compute ξ_6. The equation (3.6) shows the data elements needed to determine ξ_6. The processors on which these elements reside originally are indicated in subscripts, and the elements of t and s in which these terms are collected are highlighted in Fig-

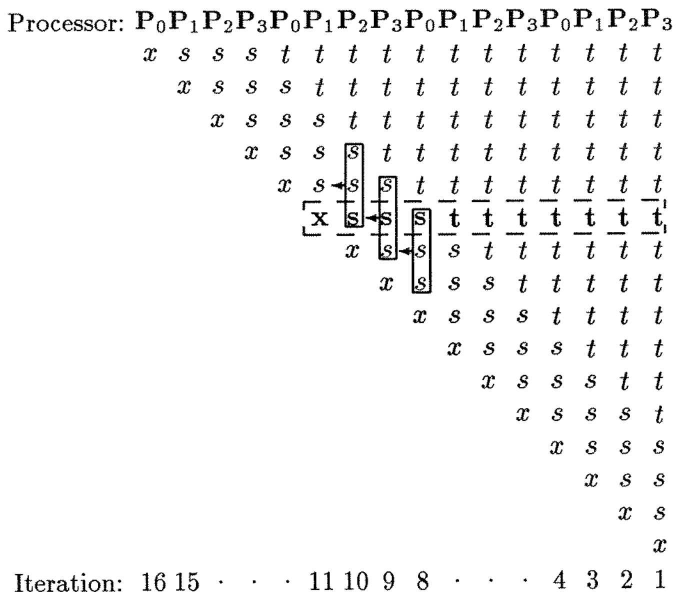
Processor: P0P1P2P3P0P1P2P3P0P1P2P3P0P1P2P3
Iteration: 16 15 · · · 11 10 9 8 · · · 4 3 2 1

Figure 3.16: Data Flow During Parallel Back-Substitution

ure 3.16.

$$\begin{aligned}\xi_6 \;=\; & \Big[(\beta_6 - v_{6,16}\xi_{16} - v_{6,12}\xi_{12} - v_{6,8}\xi_8)_{\mathbf{P}_3} \\ & +(-v_{6,15}\xi_{15} - v_{6,11}\xi_{11} - v_{6,7}\xi_7)_{\mathbf{P}_2} \\ & +(-v_{6,14}\xi_{14} - v_{6,10}\xi_{10})_{\mathbf{P}_1} \\ & +(-v_{6,13}\xi_{13} - v_{6,9}\xi_9)_{\mathbf{P}_0}\Big] / (v_{6,6})_{\mathbf{P}_1} \qquad (3.6)\end{aligned}$$

In step 8, the terms $(-v_{6,13}\xi_{13} - v_{6,9}\xi_9)$ are shifted into σ_3 from t on $\mathbf{P}_0$ and then sent to $\mathbf{P}_3$ with the rest of vector s. These terms were accumulated in t during steps 4 and 8. In step 9, the sum $(\beta_6 - v_{6,16}\xi_{16} - v_{6,12}\xi_{12} - v_{6,8}\xi_8)$ is added to σ_3 and the sum stored in σ_2 on $\mathbf{P}_3$. The vector s is then sent on to $\mathbf{P}_2$ which in step 10, adds the terms $(-v_{6,15}\xi_{15} - v_{6,11}\xi_{11} - v_{6,7}\xi_7)$ to σ_2 and stores the sum in σ_1. Finally, in step 11, $\mathbf{P}_1$ receives s from $\mathbf{P}_2$ and the terms $(-v_{6,14}\xi_{14} - v_{6,10}\xi_{10})$ are added to σ_1 on $\mathbf{P}_1$ and divided by $v_{6,6}$ to yield ξ_6.

We begin the complexity analysis by making the simplifying assumption that the number of processors evenly divides the number of columns in the matrix (i.e., $n = mp$ where m is an integer). It is conceptually helpful to group iterations into m cycles of p iterations each, and to consider the passage of the communications buffer s around the ring of processors as the algorithm loops through each cycle. Each cycle then corresponds to the passage of s once around the ring of processors. In the first cycle, the elements $\xi_n, \ldots, \xi_{n-p+1}$ are determined, and in the second cycle, the elements $\xi_{n-p}, \ldots, \xi_{n-2p+1}$ are determined and so on until $\xi_p, \ldots, \xi_1$ are determined in the mth cycle. Each cycle (except the first) begins with $\mathbf{P}_{p-1} = \mathbf{P}(n)$ receiving the vector s, determining the current element of x and then updating and sending the vector s to $\mathbf{P}_{p-2}$ before updating the vector t. Each cycle (except the last) ends when $\mathbf{P}_0$ sends s to $\mathbf{P}_{p-1}$ for it to begin the next cycle of iterations. When s arrives at $\mathbf{P}_{p-1}$, this processor may begin using and updating the vector immediately or it may have to finish updating the vector t from its previous iteration before initiating the next cycle of iterations.

The buffer, s, must pass completely around the ring exactly $(m-1)$ times and then in the mth pass it travels all but the last link of the ring. The time complexity of the algorithm is the time for s to circle the ring $(m-1)$ times plus time for the last partial circuit plus any delays between cycles. During a complete pass around the ring each processor performs p floating point operations and then sends the buffer to its neighbor. This takes time equal to $p\gamma + \alpha + p\beta$, making the time for a complete cycle, $p^2\gamma + p(\alpha + p\beta)$. The time for s to make $(m-1)$ complete cycles is then

$$\begin{aligned} \mathrm{T}_{complete} &= (m-1)(p^2\gamma + p(\alpha + p\beta)) \\ &= (n-p)(\alpha + p\beta + p\gamma). \end{aligned}$$

The time to complete the last partial circuit must account for the shrinking of the vector s in each iteration and is given by,

$$\begin{aligned} \mathrm{T}_{partial} &= \left(\sum_{j=0}^{p-1} (p-j)\gamma \right) + (p-1)(\alpha + p\beta) \\ &= \frac{p(p+1)}{2}\gamma + (p-1)(\alpha + p\beta). \end{aligned}$$

Li and Coleman show that only $\mathbf{P}(n) = \mathbf{P}_{p-1}$ can delay s as it is passed around the ring of processors. It suffices to consider only this processor when determining the total time that s is delayed. In the ith cycle $(i = 1, \ldots, m)$, $\mathbf{P}(n)$ performs $(n - ip)$ flops to update the remainder of t. The time for s to circle the ring and return to the sender is $p(\alpha + p\beta) + p(p-1)\gamma$. Whenever the time for $\mathbf{P}(n)$ to

p	$n_{delay}(p)$
2	79
4	180
8	449
16	1491
24	2680

Table 3.1: Column index below which delay is zero

update t exceeds the time for s to return to $\mathbf{P}(n)$, the vector s is delayed. Therefore the start of the $(i+1)$th cycle is delayed whenever

$$(n - ip)\gamma > p(\alpha + p\beta) + p(p-1)\gamma,$$

or when

$$i < k \qquad \text{where } k = \left(\frac{n}{p} - \frac{\alpha + p\beta}{\gamma} - (p-1)\right),$$

The column index below which the delay is zero is a function of p, (see Table 3.1) and is found by letting $i = k - 1$,

$$n_{delay}(p) = n - p(k-1) = \frac{p(\alpha + p\beta)}{\gamma} + p^2.$$

The entries in Table 3.1 can also be interpreted as being the problem size above which communications is almost completely overlapped by computation.

If the problem size is less than or equal to $n_{delay}(p)$, then no delay occurs and the time complexity is,

$$\begin{aligned} \mathrm{T}_p(n) &= \mathrm{T}_{complete} + \mathrm{T}_{partial} & n \le n_{delay}(p) \\ &= (n-1)(\alpha + p\beta) + \left(n - \frac{p-1}{2}\right) p\gamma. & (3.7) \end{aligned}$$

If the problem size is greater than $n_{delay}(p)$ then a delay occurs and the delay is $d_i = (n - ip)\gamma - p(\alpha + p\beta) - p(p-1)\gamma$. Using the approximation that $\lceil k \rceil - 1 \approx k$ the total time that s is delayed is,

$$\begin{aligned} \mathrm{T}_{delay} &\approx \sum_{i=1}^{k} d_i \\ &= \left(\sum_{i=1}^{k} (n - ip)\gamma\right) - k(p(\alpha + p\beta) + p(p-1)\gamma) \\ &= \frac{1}{2} p\gamma k(k-1). \end{aligned}$$

For problems larger than $n_{delay}(p)$ the time complexity is,

$$\begin{aligned} \mathrm{T}_p &= \mathrm{T}_{complete} + \mathrm{T}_{partial} + \mathrm{T}_{delay} && n > n_{delay}(p) \\ &\approx \left(\frac{pk^2}{2} + np - \frac{p^2}{2}\right)\gamma + (n-1)(\alpha + p\beta). \end{aligned}$$

Noting that $k \approx n/p$ for large n and ignoring low order terms we get

$$\mathrm{T}_p(n) \approx \left(\frac{n^2}{2p}\right)\gamma + (n-1)(\alpha + p\beta) \qquad n > n_{delay}(p). \tag{3.8}$$

For a fixed value of p, as the problem size becomes very large we have

$$\lim_{n\to\infty} \frac{\mathrm{T}_1(n)}{\mathrm{T}_p(n)} = p,$$

so the speedup approaches p and efficiency nears 100%, but as indicated in Table 3.1 these values are prohibitively large for even a moderate number of processors.

3.3.3 Implementation and Numerical Experiments

The routine PTRSL (**p**arallel **tr**iangular **s**o**l**ve) drives each node in a ring of processors to solve the upper triangular system of equations $Ax = b$ by back substitution. The calling sequence and argument descriptions are given in Figure 3.17.

The observed execution times for two, four and sixteen processors to solve problems ranging from $n = 10$ to $n = 500$ are plotted in Figure 3.18. The transition from linear to quadratic behavior is apparent for two and four processors. This transition occurs when the problem size becomes large enough for T_{delay} to become non-zero. In the case of sixteen processors this requires n to be greater than 1400 and hence the timings remain linear in the range of problem sizes tested (see Table 3.1).

Figure 3.19 exhibits the agreement between observed and theoretical run times. Again, discrepancies between these times are explained by the combined effects of several factors. Figure 3.20 plots the observed efficiencies attained by the parallel algorithm for various ring and problem sizes. Reasonable efficiencies are attained for two and four processors once the problem size has grown large enough to mask the expense of sending the buffer around the ring with the useful computation of updating the t vector.

PTRSL(N,A,X,T,S,INFO)

- On Entry

  N is the order of the matrix to be factored.

  A is a pointer to a vector of type FLT which contains columns of the upper triangular matrix.

  T is a pointer to a vector of type FLT which contains the first $(n-p+1)$ elements of b on $\mathbf{P}(n)$ and 0 on all other processors.

  S is a pointer to a vector of type FLT which contains the last $(p-1)$ elements of b on $\mathbf{P}(n)$ and 0 on all other processors.

- On Return

  A is a pointer to a vector of type FLT which contains columns of the unchanged upper triangular matrix.

  X is a pointer to a vector of type FLT which contains the solution. The full solution vector is distributed among the processors such that $\xi_j \in \mathbf{P}(j)$.

  INFO is a pointer to an integer, which has value 0 if the matrix A is non-singular or has value k if the kth diagonal element of A is zero.

Figure 3.17: PTRSL

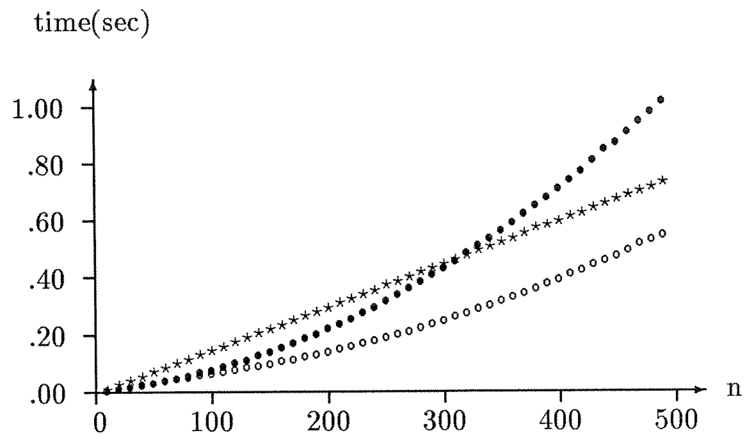

Figure 3.18: Observed timings attained during backward substitution (key:: •: $p = 2$, ∘: $p = 4$, ⋆: $p = 16$)

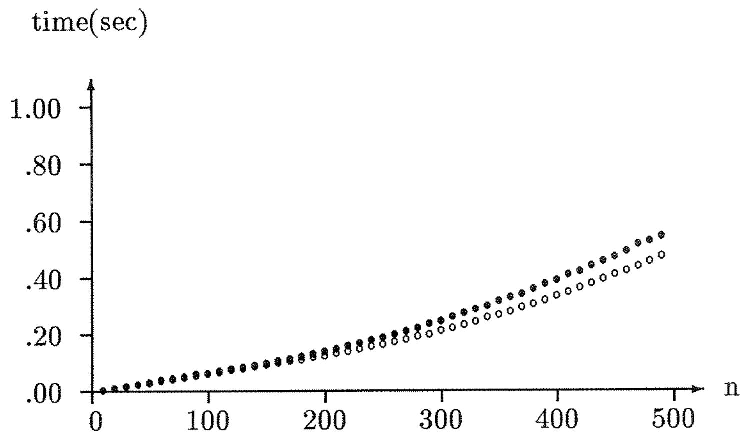

Figure 3.19: Observed and expected timings for $p = 4$ during backward substitution (key:: •: observed, ∘: expected)

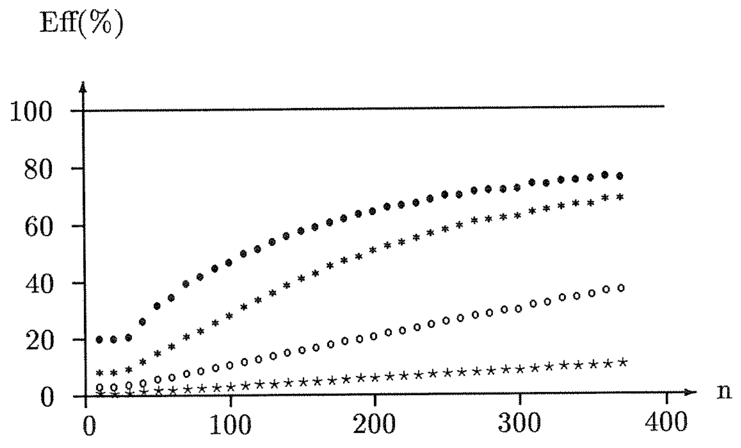

Figure 3.20: Observed efficiencies attained during backward substitution (key:: •: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$)

Chapter 4

Q-R Factorization

One application of the algorithms in this chapter is concerned with solving the least squares problem of minimizing $||Ax - b||_2$, where $A \in \mathbf{R}^{m\times n}$ for $m > n$ and $b \in \mathbf{R}^m$. We will examine the Householder and Gram-Schmidt methods for transforming this problem into an equivalent yet easier problem to solve. Householder orthogonalization factors A into the product of a unitary matrix Q and an upper trapezoidal matrix R. The Gram-Schmidt method factors A into the product of a matrix $Q \in \mathbf{R}^{m\times n}$ with orthonormal columns and $R \in \mathbf{R}^{n\times n}$, an upper triangular matrix with positive diagonal elements.

Once computed we can use this QR factorization of A to transform the least squares problem as follows,

$$Q^T A = R = \left[\begin{array}{c} R_1 \\ 0 \end{array} \right] \begin{array}{l} n \\ m-n \end{array}$$

where R_1 is upper triangular and

$$Q^T b = \left[\begin{array}{c} c \\ d \end{array} \right] \begin{array}{l} n \\ m-n \end{array} .$$

The least squares problem is then to minimize,

$$\begin{aligned} ||Ax - b||_2^2 &= ||Q^T Ax - Q^T b||_2^2 \\ &= ||R_1 x - c||_2^2 + ||d||_2^2 \end{aligned}$$

for any $x \in \mathbf{R}^n$. If $\text{rank}(A) = \text{rank}(R_1) = n$ then a unique x_{LS} which minimizes $||Ax - b||_2$ exists and is defined as the solution to the upper triangular system,

$$R_1 x_{LS} = c.$$

4.1 Householder Orthogonalization

A Householder transformation or elementary reflector has the form

$$H = I - \beta vv^T,$$

where $\beta = 2/v^Tv$. Householder matrices are symmetric, orthogonal and involutory, and can be used to introduce a contiguous block of zeros into a vector. In fact if $x \in \mathbf{R}^n$, the Householder transformation,

$$H = I - 2\frac{vv^T}{v^Tv}$$

where

$$v = x \pm \|x\|_2\, e_1,$$

has the property that

$$Hx = \mp\|x\|_2\, e_1.$$

The vector v is chosen to be

$$v = x + \text{sign}(\xi_1)\|x\|_2\, e_1,$$

to avoid introducing large relative error in the factor $\beta = 2/v^Tv$ when x is close to a multiple of e_1. This guarantees nearly perfect orthogonality in the computed H [2].

4.1.1 Sequential Algorithm

This algorithm (see Figure 4.1) factors $A \in \mathbf{R}^{m\times n}$ into the product QR where $Q \in \mathbf{R}^{m\times m}$ is unitary and $R \in \mathbf{R}^{m\times n}$ is upper trapezoidal, by applying Householder transformations to introduce zeros below the diagonal of A.

The algorithm begins with $A = A^{(1)} = (a_1^{(1)}, a_2^{(1)}, \ldots, a_n^{(1)})$ and proceeds by first determining a Householder matrix, H_1, such that

$$H_1 a_1^{(1)} = -\text{sign}(\alpha_{11}^{(1)})\|a_1^{(1)}\|_2\, e_1$$

and applying it to form

$$A^{(2)} = H_1 A^{(1)}.$$

Algorithm 4.1 *The algorithm below uses Householder transformations to overwrite A with its QR decomposition. On return the upper triangle of A contains the nonzero elements of R. The vectors v and constants β which define the Householder matrices that were used at each step are stored below the diagonal of A.*

```
$A^{(1)} = A$
for $k = 1, \ldots, n$
    determine $\tilde{H}_k$ such that $\tilde{H}_k(\alpha_{kk}, \ldots, \alpha_{mk})^T = (\rho_{kk}, 0, \ldots, 0)^T$
    $A^{(k+1)} \leftarrow$ diag$(I_{k-1}, \tilde{H}_k)A^{(k)}$
```

Figure 4.1: Householder Orthogonalization

This sequence is repeated n times. During the kth step, $A^{(k)}$ has the form

$$A^{(k)} = \begin{array}{cc} \begin{array}{ccc} k-1 & 1 & n-k \end{array} & \\ \left(\begin{array}{ccc} R_k & r_k & B_k \\ 0 & c_k & D_k \end{array}\right) & \begin{array}{l} k-1 \\ m-k+1 \end{array} \end{array},$$

where $R_k \in \mathbf{R}^{(k-1)\times(k-1)}$ is upper triangular. We now determine the Householder matrix $\tilde{H}_k \in \mathbf{R}^{(m-k+1)\times(m-k+1)}$ such that $\tilde{H}_k c_k = \rho_{kk} e_1$. Then we set $H_k = \text{diag}(I_{k-1}, \tilde{H}_k)$ and set $A^{(k+1)} = H_k A^{(k)}$.

$$\begin{aligned} A^{(k+1)} &= \begin{array}{cc} \begin{array}{cc} k & n-k \end{array} & \\ \left(\begin{array}{cc} \left(\begin{array}{cc} R_k & r_k \\ 0 & \rho_{kk} \end{array}\right) & B_k \\ 0 & \tilde{H}_k D_k \end{array}\right) & \begin{array}{l} k \\ m-k \end{array} \end{array} \\ &= \begin{array}{cc} \begin{array}{ccc} k & 1 & n-k-1 \end{array} & \\ \left(\begin{array}{ccc} R_{k+1} & r_{k+1} & B_{k+1} \\ 0 & c_{k+1} & D_{k+1} \end{array}\right) & \begin{array}{l} k \\ m-k \end{array} \end{array} \end{aligned}$$

Note that in forming $A^{(k+1)}$ we need only compute $\tilde{H}_k D_k$.

As seen above, the Householder matrix to transform the vector x such that $Hx = -\text{sign}(\xi_1)\|x\|_2\, e_1$, is completely determined by $v = x + \text{sign}(\xi_1)\|x\|_2\, e_1$ and $\beta = 2/v^T v$. Rather than store $Q = H_1 H_2 \ldots H_n$ explicitly, we can store Q in factored form by saving v and β from each H_k. The cost of determining H_k is approximately $(m-k+1)\gamma$. In forming the product $\tilde{H}_k D_k = (I - \beta v v^T)(d_1, \ldots, d_{n-k})$ we need not form $\tilde{H}_k$ explicitly. This update can be computed as $d_i \leftarrow d_i - \beta(v^T d_i)v$. The cost

Algorithm 4.2 *This algorithm drives processor $\mathbf{P}_i$, a node in a ring of processors to form the orthogonal QR decomposition of a general rectangular matrix, A, by applying Householder transformations. The matrix is distributed among the processors in a column wrapped fashion.*

```
for $k = 1, \ldots, n$
  if $k \in \mathbf{P}_i$
    determine $\tilde{H}_k$ : $\tilde{H}_k(\alpha_{kk}, \ldots, \alpha_{mk})^T = \rho_{kk}\, e_1$   $[(m-k+1)\gamma]$
    broadcast $v$ and $\beta$                                   $[2(\alpha + (m-k)\beta)]$
  else
    receive $v$ and $\beta$
  update $d_j \leftarrow d_j - \beta(v^T d_j)v$,  $j \in \mathbf{P}_i > k$              $[(2\lceil \frac{n-k}{p} \rceil (m-k+1))\gamma]$
```

Figure 4.2: Parallel Householder Orthogonalization

of updating this submatrix is approximately $2(n-k)(m-k)\gamma$. The complexity of this algorithm is then,

$$T_1(m,n) \;\approx\; \sum_{k=1}^{n} 2(n-k+1)(m-k)\gamma \approx n^2 m - \frac{n^3}{3}. \qquad (4.1)$$

4.1.2 Parallel Algorithm

The parallelization of the Householder orthogonalization algorithm proceeds almost identically to that for Gaussian elimination. Again we distribute the matrix A to the ring of processors in a column wrapped fashion.

Given that each processor has approximately (n/p) columns, we consider the sequential algorithm which loops through a set of computations for each of the n columns of the matrix. Within each iteration, the sequence of computations is to determine $\tilde{H}_k$ and then apply it from the left side to the $(m-k) \times (n-k)$ submatrix D_k. The Householder transformation, $\tilde{H}_k$, can be completely determined by processor $\mathbf{P}(k)$. The submatrix, D_k, however, is distributed among all processors, so all processors are required to assist in applying the update. Once $\tilde{H}_k$ is computed, $\mathbf{P}(k)$ can broadcast v and β to the other processors and all processors can simultaneously update their portions of D_k.

The expressions in brackets, in Figure 4.2, indicate the effective contribution to the time complexity of that step in each iteration. As in the sequential algorithm, computing the Householder transformation requires $(m-k+1)\gamma$ time, and as was the case in both Gaussian elimination, and the Cholesky decomposition the effective time to perform the broadcast is equivalent to the time to perform two node-to-neighbor communications. The message here is a vector of $(m-k)$ double precision floating point numbers so the time to communicate in each iteration is $2(\alpha+(m-k)\beta)$. The $(k+1)$th iteration begins as $\mathbf{P}(k+1)$ completes updating the columns of D_k that it owns. Processor $\mathbf{P}(k+1)$ owns $\lceil(n-k)/p\rceil$ columns of this submatrix. The update of each column requires a dot product and saxpy operation each of order $(m-k)$.

Using the approximation $\lceil(n-k)/p\rceil \approx (n-k)/p+1$, the time for $\mathbf{P}(k+1)$ to update its share of the submatrix is then,

$$2\left(\frac{n-k}{p}+1\right)(m-k)\gamma.$$

Summing these three expressions over all iterations gives the total time complexity of the algorithm,

$$\begin{aligned}\mathrm{T}_p(m,n) &\approx \sum_{k=1}^{n}\Bigg[(m-k+1)\gamma+2\left(\frac{n-k}{p}+1\right)(m-k)\gamma \\ &\qquad + 2(\alpha+(m-k)\beta\Bigg] \\ &\approx \left(n^2m-\frac{n^3}{3}\right)\frac{\gamma}{p}+2n\left(m-\frac{n}{2}\right)\gamma \\ &\qquad +2n\left(\alpha+\left(m-\frac{n}{2}\right)\beta\right). \end{aligned} \tag{4.2}$$

Again, the complexity consists of a dominating term which compares favorably with the sequential time complexity, given by (4.1). For a fixed value of p, as the problem size increases we have

$$\lim_{n\to\infty}\frac{\mathrm{T}_1(n)}{\mathrm{T}_p(n)}=p,$$

so the speedup approaches p and efficiency nears 100%. The complexity also contains a term resulting from the sequential computation and a term reflecting communication costs. Both of these terms become significant and reduce efficiency as p approaches n.

PQRDC(M,N,A,INFO)

- On Entry

  M number of rows in the matrix

  N number of columns in the matrix

  A is a pointer to a vector of type FLT which contains columns of the matrix to be factored. Each column of A has been extended to a length of $m + 2$ to store further information required to recover the Householder transformation used at each step.

- On Return

  A a pointer to a vector containing columns of the matrix A which has been overwritten by R on and above the diagonal. The vectors v which define the transformation $H_k = I - \beta vv^T$ are stored below the diagonal. The first element of v and β are stored in $A[m+2][k]$ and $A[m+1][k]$ respectively. The orthogonal matrix $Q = H_1 H_2 \ldots H_n$ can be formed from these vectors.

  INFO is a pointer to an integer
  $= 0$ normal value
  $= k$ if the kth column was already reduced

Figure 4.3: PQRDC

4.1.3 Implementation and Numerical Experiments

The routine PQRDC (parallel **Q-R d**ecomposition) drives each node in a ring of processors to form the orthogonal QR decomposition of a general rectangular matrix, A, by applying Householder transformations. The matrix is distributed among the processors in a column wrapped fashion. The calling sequence and argument descriptions are given in Figure 4.3.

Figure 4.4 plots the expected and observed timings obtained by the parallel algorithm with a ring of sixteen processors on the Symult S2010 for $n = 10, 20, \ldots, 490$. For these timings, m was set to $n + 2$. Again, discrepancies between these times are explained by the combined effects of several factors, this time including the overestimation of the cost of an inner product.

Figure 4.5 plots the observed efficiency attained by the parallel algorithm for various ring sizes and problem sizes. Reasonable efficiencies can be obtained

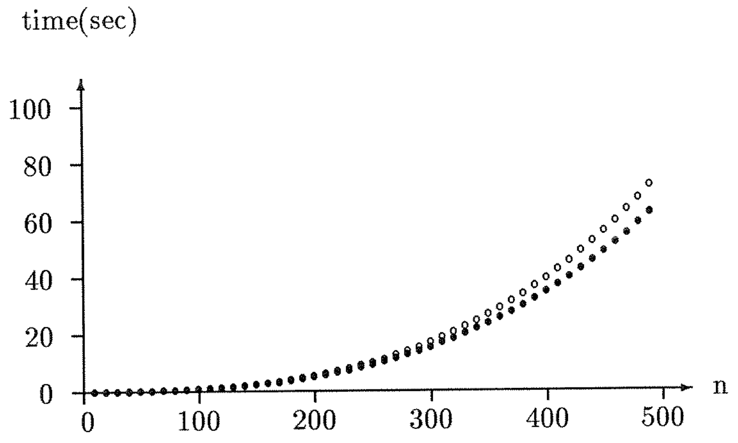

Figure 4.4: Observed and expected timings for $p = 16$ during Householder Orthogonalization (key:: •: observed, ∘: expected)

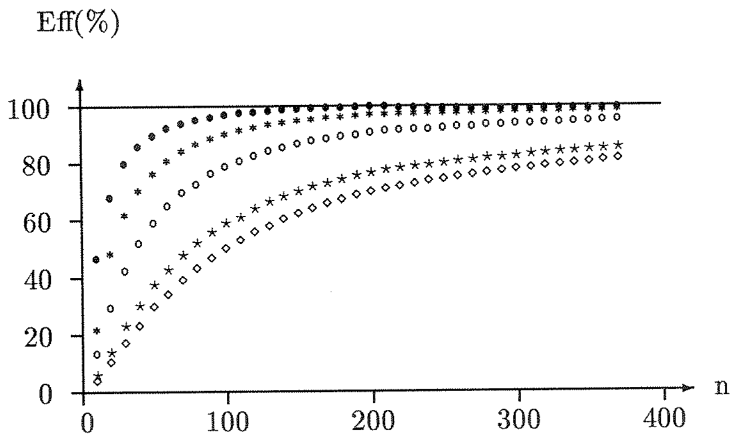

Figure 4.5: Observed efficiencies attained during Householder Orthogonalization (key:: •: $p = 2$, $*$: $p = 4$, ∘: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

once $n/p > 6$. The parallel Householder orthogonalization algorithm shows higher efficiencies for the same problem size than any of the algorithms presented in this paper. Comparing it to Gaussian elimination we see that they share the same communication expense but the computation required during each submatrix update is nearly doubled in the Q-R decomposition. This means that more useful computation is performed in parallel for the cost of a broadcast and hence the algorithm is more efficient.

4.2 Modified Gram-Schmidt Method

If we partition A and Q by columns and consider the elements of R we can express the Q-R factorization as,

$$(a_1, \ldots, a_n) = (q_1, \ldots, q_n) \begin{pmatrix} \rho_{11} & \rho_{12} & \cdots & \rho_{1n} \\ 0 & \rho_{22} & \cdots & \rho_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \rho_{nn} \end{pmatrix}$$

and clearly,

$$\begin{aligned} a_k &= \sum_{i=1}^{k} \rho_{ik} q_i \\ &= \rho_{kk} q_k + \sum_{i=1}^{k-1} \rho_{ik} q_i. \end{aligned}$$

If A has linearly independent columns, then R must be nonsingular and therefore $\rho_{kk} \neq 0$ and we can solve for q_k,

$$q_k = \frac{1}{\rho_{kk}} \left(a_k - \sum_{i=1}^{k-1} \rho_{ik} q_i \right),$$

where the orthonormality of the q_i implies that

$$\rho_{ik} = q_i^T a_k \qquad \text{for } i = 1, \ldots, k.$$

The expression for q_k can be interpreted as the normalized vector resulting from subtracting from a_k all its components in the direction of the already formulated q_i. This ensures that q_k is orthogonal to span$(q_1, \ldots, q_{k-1})$.

These expressions for q_k and ρ_{ik} lead to the Classical Gram-Schmidt (CGS) method which generates the kth column of Q and R in the kth step of the algorithm, but unfortunately exhibits poor numerical behavior [2]. However a rearrangement of the computations yields the more stable Modified Gram-Schmidt (MGS) method.

4.2.1 Sequential Algorithm

Since Q has orthonormal columns, $Q^TQ = I$ and we can rewrite $A = QR$ as $Q^TA = R$ or,

$$\begin{pmatrix} q_1^T \\ q_2^T \\ \vdots \\ q_n^T \end{pmatrix} (a_1, \ldots, a_n) = \begin{pmatrix} \rho_{11} & \rho_{12} & \cdots & \rho_{1n} \\ 0 & \rho_{22} & \cdots & \rho_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & \rho_{nn} \end{pmatrix}.$$

From this we see that,

$$\rho_{kj} = q_k^T a_j \qquad \text{for } j = k, \ldots, n$$

and if we normalize q_k by setting $q_k = a_k/\|a_k\|_2$, then

$$\rho_{kk} = q_k^T a_k = \|a_k\|_2.$$

The condition that the q_i be orthogonal requires that we subtract from a_j its component in the direction of q_k, for each a_j not yet orthogonal to q_k (i.e., $j > k$). This can be done each time a new q_k is formed by,

$$a_j \leftarrow a_j - (q_k^T a_j) q_k \qquad \text{for } j > k.$$

The algorithm (see Figure 4.6) begins by setting $q_1 = a_1/\|a_1\|_2$ and then determining the first row of R and subtracting the q_1 component from each of $a_2, a_3, \ldots, a_n$. In the second iteration, q_2 is normalized the second row of R determined and the q_2 component subtracted from $a_3, \ldots, a_n$. At this point, $q_2 = 1/\rho_{22}(a_2 - (q_1^T a_2) q_1)$ and the orthogonality of q_1 and q_2 is easily verified since $q_1^T q_1 = 1$ and

$$q_1^T q_2 = \frac{1}{\rho_{22}}(q_1^T a_2 - (q_1^T a_2) q_1^T q_1) = 0.$$

The orthogonality of the q_i can be shown inductively.

In the kth iteration finding the norm and scaling a_k requires approximately $2m\gamma$ time and the execution of the inner loop takes approximately $2(n-k)m\gamma$ time. Summing over all iterations the total time complexity is given as,

$$\mathrm{T}_1(m, n) \approx \sum_{k=1}^{n} 2m(n-k+1)\gamma \approx mn^2\gamma. \tag{4.3}$$

Algorithm 4.3 *This algorithm uses MGS to factor A into the product QR where A is overwritten by Q. In the kth iteration the kth column of Q and the kth row of R are determined.*

```
for $k = 1, \ldots, n$
    $\rho_{kk} \leftarrow \|a_k\|_2$
    $q_k = a_k \leftarrow a_k / \rho_{kk}$
    for $j = k+1, \ldots, n$
        $\rho_{kj} \leftarrow q_k^T a_j$
        $a_j \leftarrow a_j - \rho_{kj} q_k$
```

Figure 4.6: Modified Gram-Schmidt

4.2.2 Parallel Algorithm

The parallel MGS algorithm (see Figure 4.7) employs only the broadcast primative to factor a matrix A with linearly independent columns into the product of a matrix Q with orthonormal columns and an upper triangular matrix R. Again we distribute the matrix A to the ring of processors in a column wrapped fashion so that each processor has approximately (n/p) columns.

The algorithm loops through a set of computations for each of the n columns of the matrix A. In the kth iteration, the processor that owns a_k, $\mathbf{P}(k)$, generates q_k by normalizing a_k and then broadcasts this vector to all other processors. Once a processor receives q_k it can update its columns a_j $(j > k)$ by subtracting the q_k component of a_j from a_j.

In Figure 4.7, the expressions in brackets indicate the effective contribution to the time complexity of that step in each iteration. Determining ρ_{kk} and scaling a_k requires $2m\gamma$ time, and the effective time to perform the broadcast is equivalent to the time to perform two node-to-neighbor communications with a message of m double precision floating point numbers. Communication time in each iteration is $2(\alpha + m\beta)$. The $(k+1)$th iteration begins as $\mathbf{P}(k+1)$ completes updating its columns of A. Processor $\mathbf{P}(k+1)$ owns $\lceil (n-k)/p \rceil$ columns that must be updated. The update of each column requires a dot product and saxpy operation each of order m. Using the approximation $\lceil (n-k)/p \rceil \approx (n-k)/p + 1$, the time for $\mathbf{P}(k+1)$ to update its columns is then, $2m(\frac{n-k}{p} + 1)\gamma$. Summing these three expressions over all

Algorithm 4.4 *This algorithm drives $\mathbf{P}_i$, a node in a ring of processors to form the orthonormal basis which spans the column space of A. A is overwritten by Q of the product $A = QR$, where Q has orthonormal columns and R is an upper triangular matrix with positive diagonal elements. The matrix is distributed among the processors in a column wrapped fashion.*

```
for $k = 1, \ldots, n$
    if $k \in \mathbf{P}_i$
        $\rho_{kk} \leftarrow \|a_k\|_2$                          $[m\gamma]$
        $q_k = a_k \leftarrow a_k / \rho_{kk}$                   $[m\gamma]$
        broadcast $q_k$                           $[2(\alpha + m\beta)]$
    else
        receive $q_k$
    for $j \in \mathbf{P}_i$ and $j > k$
        $\rho_{kj} \leftarrow q_k^T a_j$
        $a_j \leftarrow a_j - \rho_{kj} q_k$                     $[2m\lceil \frac{n-k}{p} \rceil \gamma]$
```

Figure 4.7: Parallel Modified Gram-Schmidt

iterations gives the total time complexity of the algorithm,

$$\begin{aligned} \mathrm{T}_p(n) &\approx \sum_{k=1}^{n} \left[2m\gamma \left(\frac{n-k}{p} + 2 \right) + 2(\alpha + m\beta) \right] \\ &\approx \left(\frac{n^2 m}{p} + 4nm \right) \gamma + 2n(\alpha + m\beta). \end{aligned} \tag{4.4}$$

This algorithm also approaches 100% efficiency for large n and fixed p. Comparing (4.3) and (4.4) we see that,

$$\lim_{n \to \infty} \frac{\mathrm{T}_1(n)}{\mathrm{T}_p(n)} = p.$$

However, as p approaches n the two terms $(2nm)\beta$ and $(4nm)\gamma$ gain significance and efficiency drops. The term $(4nm)\gamma$ reflects the cost of the sequential portion of each iteration, where one processor determines ρ_{kk} and normalizes q_k.

4.2.3 Implementation and Numerical Experiments

The routine PMGS (parallel modified Gram-Schmidt) drives each node in a ring of processors to form the orthonormal basis which spans the column space

PMGS(M,N,A,R,INFO)

- <u>On Entry</u>

  M number of rows in the matrix

  N number of columns in the matrix

  A is a pointer to a vector of type FLT which contains columns of the matrix to be factored.

- <u>On Return</u>

  A a pointer to a vector containing columns of the matrix A which has been overwritten by Q a matrix with orthonormal columns.

  R a pointer to a vector containing columns of the $n \times n$ upper triangular matrix R, distributed among the processors in a column wrapped fashion.

  INFO is a pointer to an integer
  $= 0$ normal value
  $= k$ if the kth column was linearly dependent

Figure 4.8: PMGS

of A, a rectangular matrix with linearly independent columns. A is factored into the product QR, where Q has orthonormal columns and R is an upper triangular matrix with positive diagonal elements. A is overwritten by Q. The matrix is distributed among the processors in a column wrapped fashion. The calling sequence and argument descriptions are given in Figure 4.8.

Figure 4.9 plots the observed timings obtained by PMGS with various ring sizes on the Symult S2010 for problems ranging up to $n = 490$. For these timings m was set to $n + 2$. Because of memory restrictions, the maximum problem size was limited below 490 for certain ring sizes. Figure 4.10 exhibits the agreement between observed and theoretical run times.

Figure 4.11 plots the observed efficiency attained by the parallel algorithm for various ring sizes and problem sizes. Reasonable efficiencies can be obtained once $n/p > 6$. The parallel modified Gram-Schmidt algorithm shows similar efficiencies to the parallel Householder orthogonalization algorithm. Each communication is more expensive yet more computation is performed for each broadcast.

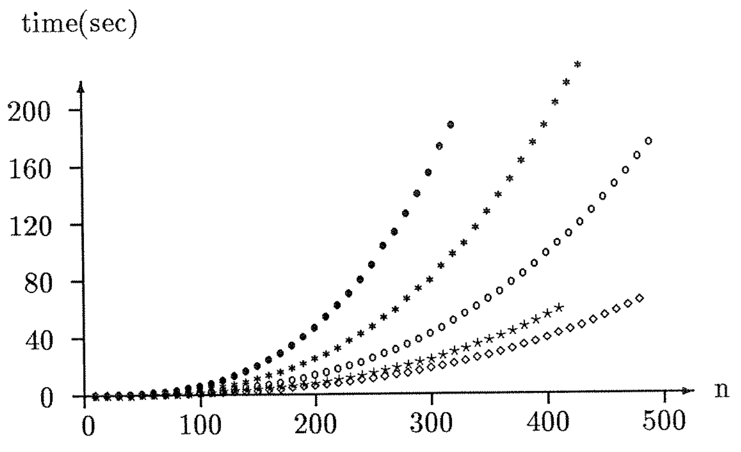

Figure 4.9: Observed timings attained during Modified Gram-Schmidt (key:: •: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

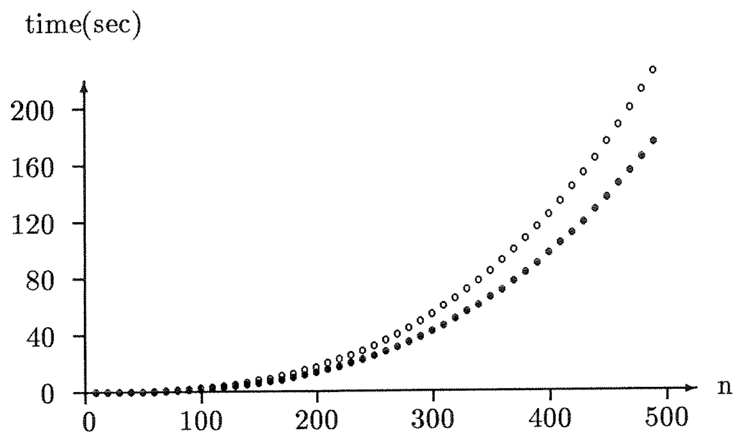

Figure 4.10: Observed and expected timings for $p = 8$ during Modified Gram-Schmidt (key:: •: observed, $\circ$: expected)

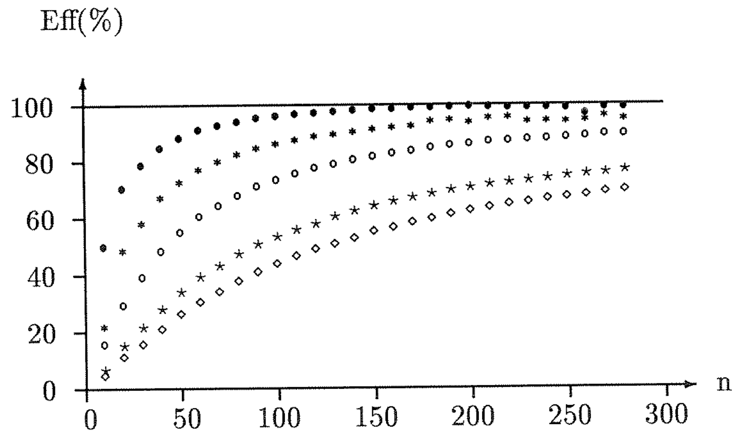

Figure 4.11: Observed efficiency attained during Modified Gram-Schmidt (key:: $\bullet$: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

Chapter 5

Householder Reduction to Hessenberg Form

In this chapter we will describe the reduction of a matrix to upper Hessenberg form by orthogonal similarity transformations. If the matrix is symmetric the result is a tridiagonal matrix, however the algorithm that will be presented does not expect or take advantage of symmetry.

A matrix, A, is in upper Hessenberg form if $\alpha_{ij} = 0$ for $i > j+1$. A 5×5 upper Hessenberg matrix has the form,

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

Reducing a matrix to upper Hessenberg form, before applying the QR algorithm to compute the Schur decomposition, lowers the cost of an iteration by an order of magnitude from $O(n^3)$ to $O(n^2)$.

5.1 Sequential Algorithm

The object of the algorithm is to compute the product,

$$U^T A U = H$$

where H is upper Hessenberg and U is unitary and is the product of Householder matrices, $H_1, \ldots, H_{n-2}$.

The algorithm (see Figure 5.1) begins by determining a Householder transformation, $\tilde{H}_1$ to zero the first column of $A = A^{(1)}$ below the first subdiagonal. A is then multiplied by

$$H_1 = \begin{pmatrix} I_1 & 0 \\ 0 & \tilde{H}_1 \end{pmatrix}$$

from the left and the right. Post multiplication of $H_1A^{(1)}$ by H_1 does not affect the first column so we have,

$$A^{(2)} = H_1A^{(1)}H_1 = \begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{pmatrix}.$$

In the next step $H_2 = \operatorname{diag}(I_2, \tilde{H}_2)$ is determined so that,

$$A^{(3)} = H_2A^{(2)}H_2 = \begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{pmatrix}.$$

Again the application of H_2 from the right does not fill in the zero elements in the first or second columns.

More generally, let $A^{(1)} = A \in \mathbf{R}^{n\times n}$ and after the $(k-1)$th step

$$(H_1, \ldots, H_{k-1})^T A (H_1, \ldots, H_{k-1}) = A^{(k)} = \begin{array}{c} \left(\begin{array}{ccc} \multicolumn{2}{c}{A_{11}^{(k)}} & A_{12}^{(k)} \\ 0 & a_{22}^{(k)} & A_{23}^{(k)} \end{array}\right) \begin{array}{l} k \\ n-k \end{array} \\ \begin{array}{ccc} k-1 & 1 & n-k \end{array} \end{array}$$

where $A_{11}^{(k)} \in \mathbf{R}^{k\times k}$ is upper Hessenberg. In the kth step we determine $\tilde{H}_k \in \mathbf{R}^{(n-k)\times(n-k)}$ such that $\tilde{H}_k a_{22}^{(k)}$ is a multiple of $e_1 \in \mathbf{R}^{n-k}$ (cf. Section 4.1.1) and compute $H_kA^{(k)}H_k$, where $H_k = \operatorname{diag}(I_k, \tilde{H}_k)$. The premultiplication in the kth iteration takes the form,

$$\begin{aligned} H_kA^{(k)} &= \begin{pmatrix} I_k & 0 \\ 0 & \tilde{H}_k \end{pmatrix} \begin{pmatrix} \multicolumn{2}{c}{A_{11}^{(k)}} & A_{12}^{(k)} \\ 0 & a_{22}^{(k)} & A_{23}^{(k)} \end{pmatrix} \\ &= \begin{pmatrix} \multicolumn{2}{c}{A_{11}^{(k)}} & A_{12}^{(k)} \\ 0 & \eta e_1 & \tilde{H}_k A_{23}^{(k)} \end{pmatrix} \end{aligned}$$

Post multiplication by H_k produces,

$$\begin{aligned} A^{(k+1)} = H_kA^{(k)}H_k &= \begin{pmatrix} A_{11}^{(k+1)} & A_{12}^{(k)} \\ 0 & \tilde{A}_{23}^{(k)} \end{pmatrix} \begin{pmatrix} I_k & 0 \\ 0 & \tilde{H}_k \end{pmatrix} \\ &= \begin{pmatrix} A_{11}^{(k+1)} & A_{12}^{(k)}\tilde{H}_k \\ 0 & \tilde{A}_{23}^{(k)}\tilde{H}_k \end{pmatrix}. \end{aligned}$$

Algorithm 5.1 *This algorithm overwrites $A \in \mathbf{R}^{n\times n}$ with the upper Hessenberg matrix U^TAU where $U = H_1 \ldots H_{n-2}$ is the product of Householder matrices. The vector v and scalar $\beta = 2/v^Tv$ which determine $\tilde{H}_k$ are stored below the first subdiagonal of A.*

```
$A^{(1)} = A$
for $k = 1, \ldots, n-2$
    determine $\tilde{H}_k$ such that $\tilde{H}_k(\alpha_{k+1,k}, \ldots, \alpha_{nk})^T = (\eta, 0, \ldots, 0)^T$
    $\tilde{H}_k = $ diag$(I_k, \tilde{H}_k)$
    $A^{(k+1)} \leftarrow H_k A^{(k)} H_k$
```

Figure 5.1: Householder Reduction to Hessenberg Form

The cost of determining $\tilde{H}_k$ is $(n-k)\gamma$, and the premultiplication which consists of computing $\tilde{A}_{23}^{(k)} = \tilde{H}_k A_{23}^{(k)}$ requires $2(n-k)^2\gamma$ time, as explained in Section 4.1.1. The post multiplication consists of computing both $A_{12}^{(k)}\tilde{H}_k$ and $\tilde{A}_{23}^{(k)}\tilde{H}_k$, and requires $2n(n-k)\gamma$ time. To see this let us examine how $A_{12}^{(k)}\tilde{H}_k$ can be formed. Let $A_{12}^{(k)} = (a_{k+1}, \ldots, a_n)$ and $\tilde{H}_k = I - \beta vv^T$, and then

$$A_{12}^{(k)}\tilde{H}_k = A_{12}^{(k)} - \beta(A_{12}^{(k)}v)v^T.$$

First we compute

$$A_{12}^{(k)}v = \sum_{j=k+1}^{n} a_j\nu_j = y$$

which takes approximately $(n-k)k\gamma$ time. Then each of the $(n-k)$ columns of $A_{12}^{(k)}\tilde{H}_k$ can be formed in $(n-k)k\gamma$ time by

$$a_j \leftarrow a_j - \beta\nu_j y.$$

Forming $A_{12}^{(k)}\tilde{H}_k$ requires $2k(n-k)\gamma$ time and by similar computations $\tilde{A}_{23}^{(k)}\tilde{H}_k$ can be formed in $2(n-k)^2\gamma$ time. Performing both of these operations completes the post multiplication. This makes the total time complexity,

$$T_1(n) = \sum_{k=1}^{n-2}\left[(n-k) + 2(n-k)^2 + 2n(n-k)\right]\gamma \approx \frac{5n^3}{3}. \qquad (5.1)$$

5.2 Parallel Algorithm

A parallel Householder reduction to upper Hessenberg form can be implemented on a ring of processors using the broadcast, vector sum, and total exchange communications primitives. Again the matrix A is distributed to the ring of processors in a column wrapped fashion so that each processor has approximately (n/p) columns.

The algorithm loops through a sequence of operations for each of the first $(n-2)$ columns of A. During the kth iteration, a Householder matrix, $\tilde{H}_k$, of order $(n-k)$ is computed, $A^{(k)}$ is multiplied by $\text{diag}(I_k, \tilde{H}_k) = H_k$ from the left and then H_k multiplies this result from the right. Up through the formation of $H_k A^{(k)}$ the algorithm is similar to the Q-R factorization by Householder transformations (cf. Section 4.1.2). The Householder transformation, $\tilde{H}_k$, can be completely determined by processor $\mathbf{P}(k)$. Once computed, $\mathbf{P}(k)$ can broadcast v and β and all processors can simultaneously participate in the premultiplication.

Post multiplication requires additional communication. The operation consists of overwritting the last $(n-k)$ columns of $A^{(k)}$ with,

$$\begin{pmatrix} A_{12}^{(k)} \\ \tilde{A}_{23}^{(k)} \end{pmatrix} \tilde{H}_k = \begin{pmatrix} A_{12}^{(k)} \\ \tilde{A}_{23}^{(k)} \end{pmatrix} - \beta \begin{pmatrix} A_{12}^{(k)} \\ \tilde{A}_{23}^{(k)} \end{pmatrix} vv^T.$$

The first step is to compute

$$y = \begin{pmatrix} A_{12}^{(k)} \\ \tilde{A}_{23}^{(k)} \end{pmatrix} v = (a_{k+1}, \ldots, a_n)v = \sum_{j=k+1}^{n} a_j \nu_j$$

which requires each processor, $\mathbf{P}_i$, to form the partial sum,

$$y^{(i)} = \sum_{a_j \in \mathbf{P}_i} a_j \nu_j$$

and then for all processors to synchronize to form and receive the vector,

$$y = \sum_i y^{(i)}.$$

This is done through the vector sum and total exchange primitives (cf. Section 2.1.4 and 2.1.6).

Once y is formed and known to each processor, all processors can complete the update of their portion of the last $(n-k)$ columns by computing,

$$a_j \leftarrow a_j - \beta \nu_j y.$$

Algorithm 5.2 *This algorithm drives processor $\mathbf{P}_i$, a node in a ring of processors, to reduce a general matrix, A, to upper Hessenberg form by applying Householder transformations. The matrix is distributed among the processors in a column wrapped fashion.*

```
for $k = 1, \ldots, n-2$
  if $k \in \mathbf{P}_i$
    determine $\tilde{H}_k : \tilde{H}_k(\alpha_{k+1,k}, \ldots, \alpha_{nk})^T = \eta e_1$   $[(n-k)\gamma]$
    broadcast $v$ and $\beta$                                  $[(p-1)(\alpha + (n-k)\beta)]$
  else
    receive $v$ and $\beta$
  update $a_j \leftarrow a_j - \beta(v^T a_j)v, \;\; j \in \mathbf{P}_i > k$             $[(2\lceil \frac{n-k}{p} \rceil (n-k))\gamma]$
  compute $y^{(i)} = \sum \nu_j a_j, \;\; j \in \mathbf{P}_i > k$                 $[\lceil \frac{n-k}{p} \rceil n\gamma]$
  vector sum $y = \sum y(i)$ leaving $y_i$ on $\mathbf{P}_i$         $[(p-1)(\alpha + \lceil \frac{n}{p} \rceil (\beta + \gamma))]$
  total exchange $y_i$                                   $[(p-1)(\alpha + \lceil \frac{n}{p} \rceil \beta]$
  update $a_j \leftarrow a_j - \beta \nu_j y, \;\; j \in \mathbf{P}_i > k$                $[\lceil \frac{n-k}{p} \rceil n\gamma]$
```

Figure 5.2: Parallel Householder Reduction to Hessenberg Form

Figure 5.2 lists the pseudo code which drives each node; the expressions in brackets indicate the effective contribution to the time complexity of that step in each iteration. As in the sequential algorithm, computing the Householder transformation requires $(n-k)\gamma$ time. Because of the subsequent synchronization required by a vector sum operation, the effective cost of the broadcast is the time for the message to completely traverse the ring. The message is a vector of $(n-k)$ double precision floating point numbers so the time to communicate in each iteration is $(p-1)(\alpha + (n-k)\beta)$. The update associated with the premultiplication requires a dot product and saxpy operation each of order $(n-k)$. Since each processor has $\lceil (n-k)/p \rceil$ columns, the time to perform this update is $2\lceil (n-k)/p \rceil (n-k)\gamma$. Computing the partial sum, $y^{(i)}$ entails scaling and summing $\lceil (n-k)/p \rceil$ vectors of length n. This costs $\lceil (n-k)/p \rceil n\gamma$ time. The vector sum with vectors of length n costs $(p-1)(\alpha + \lceil n/p \rceil \beta + \lceil n/p \rceil \gamma)$ and the total exchange of vectors of size $\lceil n/p \rceil$ costs $(p-1)(\alpha + \lceil n/p \rceil \beta)$ (cf. Section 2.1.4 and 2.1.6).

Using the approximation $\lceil x \rceil \approx x + 1$, and summing these expressions over all iterations gives the total time complexity of the algorithm,

$$\begin{aligned} \mathrm{T}_p(n) &\approx \sum_{k=1}^{n-2} \left[2\left(\frac{n-k}{p}+1\right)(2n-k)+n-k \right]\gamma \\ &\quad +(p-1)\left(2\alpha+\left(n-k+\frac{n}{p}\right)\beta\right) \\ &\approx \left(\frac{5n^3}{3p}+\frac{n^2}{2}\right)\gamma+3np\alpha+\frac{n^2p}{2}\beta. \end{aligned} \tag{5.2}$$

The term $(n^2/2)\gamma$ accounts for the cost of a single processor determining $\tilde{H}_k$ every iteration and becomes significant as p approaches n. For $n \gg p$ however, (5.2) compares favorably with the sequential time complexity (5.1) and if p is fixed and the problem size increases, 100% utilization of the nodes will be approached.

Note: Because of the synchronization required by the distributed vector sum, this algorithm cannot pipeline the computations of several iterations and mask a portion of the broadcast with computation as done in the QR decomposition. For this reason, we expect that this algorithm will perform more efficiently when implemented using the pipelined broadcast, as planned in future work.

5.3 Implementation and Numerical Experiments

The routine PGEHR (parallel general matrix Hessenberg reduction) drives each node in a ring of processors to reduce a general matrix, A, by applying Householder transformations, so that $H = U^TAU$ is upper Hessenberg and U is unitary. The matrix is distributed among the processors in a column wrapped fashion. The calling sequence and argument descriptions are given in Figure 5.3.

Figure 5.4 plots the observed timings obtained by the parallel algorithm with various ring sizes on the Symult S2010 for problems ranging up to $n = 490$. Figure 5.5 plots the expected and observed timings obtained by the parallel algorithm with a ring of sixteen processors. The almost 10% discrepancy is largely caused by the overestimation of the cost of an inner product.

Figure 5.6 plots the observed efficiency attained by the parallel algorithm for various ring sizes and problem sizes. Reasonable efficiencies can be obtained once $n/p > 10$.

PGEHR(N,A,INFO)

- <u>On Entry</u>

  N order of the matrix

  A is a pointer to a vector of type FLT which contains columns of the matrix to be reduced. Each column of A has been extended to a length of $n + 2$ to store further information required to recover the Householder transformation used at each step.

- <u>On Return</u>

  A a pointer to a vector containing columns of the matrix A which has been overwritten by H above the second subdiagonal. The vectors v which define the transformations $H_k = I - \beta vv^T$ are stored below the first subdiagonal. The first element of v and β are stored in $A[n+2][k]$ and $A[n+1][k]$ respectively. The unitary matrix $U = H_1 H_2 \ldots H_n$ can be formed from these vectors.

  INFO is a pointer to an integer
  $= 0$ normal value
  $= k$ if the kth column was already reduced

Figure 5.3: PGEHR

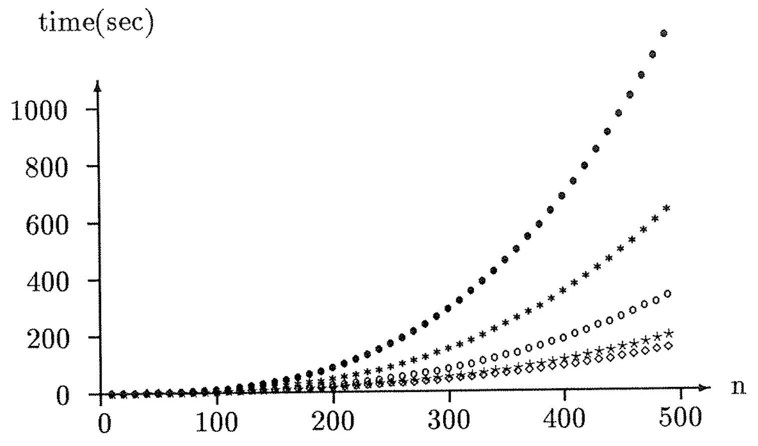

Figure 5.4: Observed timings attained during Reduction to Upper Hessenberg Form (key:: •: $p = 2$, $*$: $p = 4$, ∘: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

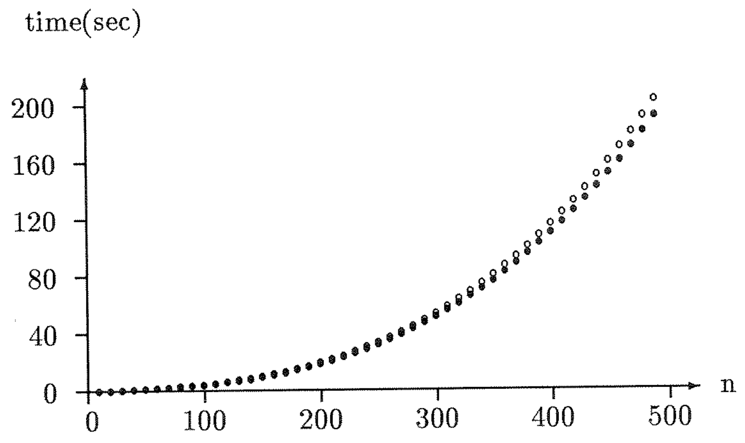

Figure 5.5: Observed and expected timings for $p = 16$ during Householder Reduction to Hessenberg Form (key:: •: observed, ∘: expected)

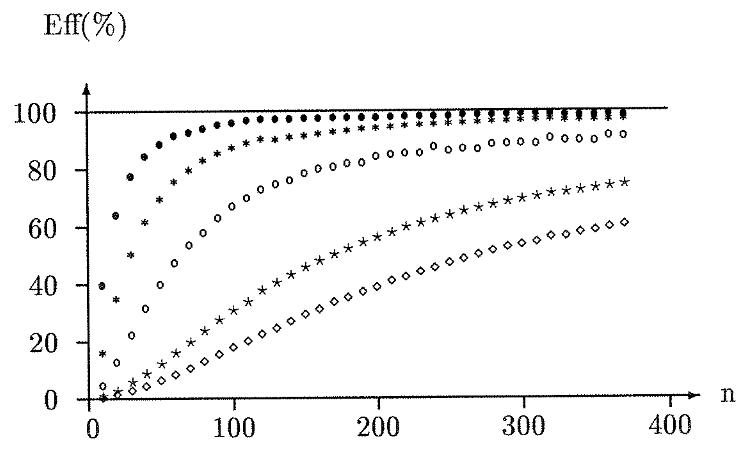

Figure 5.6: Observed efficiencies attained during Reduction to Upper Hessenberg Form (key:: $\bullet$: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

Chapter 6

Conclusion

In this chapter, we will summarize the findings of Chapters 3 through 5, and discuss other applications and the future direction of this work.

6.1 Summary

In this thesis, we have seen several matrix algorithms and their parallel implementations. Using the column wrapped storage scheme and some of the communication primatives, which were described in Chapter 2, it was shown that reasonable efficiencies can be attained in all cases for large enough problem sizes.

The communication needs, time complexity and performance for each of these algorithms are listed in Table 6.1. The column headed by n/p indicates the ratio of the problem size to the number of processors at which 70% efficiency is achieved. This ratio gives an indication of how efficiently the algorithm utilizes its communications. The smaller this ratio is, the more quickly the algorithm achieves high efficiencies as n grows.

Four of the algorithms, PGEFA, PPOFA,PQRDC and PMGS, have similar communication needs and exploit the non-pipelined implementation of the broadcast primitive by masking communication with computation (Section 3.1.2). They differ mainly in the amount of parallel and sequential computations they require. They are all comparable (within a constant factor) with respect to how quickly they achieve 70% efficiency, since for each broadcast they all perform $O(mn)$ or $O(n^2)$ flops in parallel. The algorithm PTRSL, however, only performs $O(n)$ flops per communication and reaches higher efficiencies much more slowly.

The algorithm PGEHR, has more extensive communication needs. These include a distributed vector sum and total exchange which prohibit the pipelining of computations allowed by the broadcast primitive. It is expected that this algorithm

Algorithm	Communications	Time Complexity	n/p
PGEFA	broadcast	$\left(\frac{n^3}{3p} + n^2\right)\gamma + 2n\alpha + n^2\beta$	19
PPOFA	broadcast	$\left(\frac{n^3}{6p} + \frac{3n^2}{4}\right)\gamma + 2n\alpha + n^2\beta$	29
PTRSL	node-to-neighbor	$\left(\frac{n^2}{2p}\right)\gamma + (n-1)(\alpha + p\beta)$	100
PQRDC	broadcast	$\left(n^2m - \frac{n^3}{3}\right)\frac{\gamma}{p} + 3n\left(m - \frac{n}{2}\right)\gamma$ $+ 2n\left(\alpha + \left(m - \frac{n}{2}\right)\beta\right)$	10
PMGS	broadcast	$\left(\frac{n^2m}{p} + 4nm\right)\gamma + 2n(\alpha + m\beta)$	11
PGEHR	broadcast vector sum total exchange	$\left(\frac{5n^3}{3p} + \frac{n^2}{2}\right)\gamma + 3np\alpha + \frac{n^2p}{2}\beta$	14

Table 6.1: Algorithm Scorecard:
n/p = columns per processor at which 70% efficiency was achieved.

will perform better when implemented using the pipelined broadcast, as planned in future work. However, the algorithm still achieved high efficiencies without overlapping communication and computation.

It is important to note that there are limitations to the number of processors that can be used while still achieving high efficiencies for these algorithms. First, there is the practical consideration of the computation time exceeding what is considered reasonable. For example, with $p = 1000$, the expected completion time of PQRDC is on the order of 10^4 seconds for $n/p = 10$. Secondly, for each algorithm, as p approaches n the term resulting from sequential computation and the β term in the time complexity both become more significant. Until finally, when $p = n$, communication costs are on the same order as computation costs and efficiency suffers. Note, that for the column wrapped storage scheme, p cannot exceed n.

6.2 Other Applications and Future Work

The algorithms implemented in this thesis have used only some of the communication primitives of Chapter 2. Below are some other algorithms that employ these and other primitives.

The nonsymmetric QR algorithm with Hankel-wrapped storage uses the broadcast and one-way-shift operations [13, 16]. The two-sided Jacobi method for symmetric matrices, using the blocked Hankel-wrapped storage scheme, uses the total exchange and one-way-shift operations [4, 12, 13]. The one-sided Jacobi method for finding the singular value decomposition of a matrix is parallelized using the column blocked storage scheme and the one-way-shift operations [4].

The broadcast and global compare operations are required to implement the QR decomposition with column pivoting in parallel when using column wrapped storage. Conjugate gradient methods for sparse linear systems can be parallelized using the one-way-shift and inner product operations [14]. Finally, a parallel implementation of the power method to find eigenvalues requires the vector sum and inner product.

In general, the broadcast primitive is appropriate when performing an update (e.g., scaling or applying a transformation from the left) on the columns of a matrix, which are distributed over all processors. The node-to-neighbor communication is most likely used in situations with limited parallel potential or in handling exceptions. As we have seen in algorithm 5.2, the vector sum is required when performing a matrix vector multiplication, and it must be followed by the total exchange if the result is to be distributed to all processors. The one-way-shift is useful when processors must share border data. The global compare is needed when finding pivot elements or columns. The need for the inner product is obvious and we are still searching for an application of the data transpose.

It is our goal to increase the scope of these implementations to include the standard routines of packages like LINPACK and EISPACK and to extend the primitives to parallel computers with larger numbers (> 1000) of processors.

Appendix A

Parallel Matrix Subroutines in C

A.1 PGEFA

```
#include <math.h>
extern int np;
extern int myprocno;

/*
 *   PGEFA - drives each node in a ring of p processors to
 *   factor a square general matrix, A, with type FLT(double)
 *   entries by Gaussian elimination. The matrix is distributed
 *   among the processors in a column wrapped fashion.
 *
 *   ON ENTRY
 *     a       *FLT -  pointer to vector containing columns of
 *             the matrix A to be factored.
 *     n       int - order of the matrix
 *   ON RETURN
 *     a       columns of the LU decomposition where U is upper
 *             triangular and L is unit lower triangular whose
 *             elements are the multipliers used to obtain U.
 *     ipvt    *int - pointer to vector of pivot indices
 *     info    *int - = 0 normal value
 *                    = k if the (k)th pivot element was zero
 *
 *   JULY 20,1989 - James W. Juszczak,
 *                  University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy, sswap, isamax, scopy
 *     amcom - broadcast, broad_rec
 */
```

```
pgefa(n,a,ipvt,info)
FLT *a;
int n, *ipvt, *info;
{
int pk,                         /* processor that owns col k */
    kcols,      /* # cols of kth submatrix held by processor */
    h,tmp;
register int k, i, j;
register FLT *cp, *wp, temp;
FLT *work, *col, *akk, *akj;

*info = 0;
h = (myprocno < n%np) ? n/np+1: n/np;
work = a + h*n;
for (k = 0; k < n-1; k++) {
    wp = work;
    pk = k%np;
    kcols = h-((myprocno<pk)?k/np+1:k/np);
    if (myprocno == pk) {
        col = a + (k/np)*n ;                /* top of kth col */
        akk = col + k;
        ipvt[k] = k + isamax(n-k,akk,1);   /* find pivot row */
        cp = col + ipvt[k];
        temp = *cp;                                  /* swap */
        *cp = *akk;
        *akk = temp;
        if (temp == (FLT)0.0e0) {
            *work = -1.0;
            *info = k+1;
            broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
            break;
            }
        cp = akk + 1;
        *wp++ = (FLT)ipvt[k];
        for(i=k+1;i<n;i++)             /* compute multipliers */
            *wp++ = *cp++/temp;
        broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
        scopy(n-k-1,work+1,1,akk+1,1);
        sswap(kcols-1,akk+n,n,col+ipvt[k]+n,n);
        }
    else {
        broad_rec(work);
        if (*work == -1.0) {
            *info = k+1;
            ipvt[k] = k;
            }

        else {
            ipvt[k] = (int)(*wp++);
            if ((tmp = myprocno-pk)<0) tmp += np;
            col = (a + ((k+tmp)/np)*n);       /* 1st col > k */
            sswap(kcols,col+k,n,col+ipvt[k],n);
            }
        }
    if (*work == -1.0) break;      /* rest of column is zero */

    for (j = k+1; j < n; j++)       /* update next submatrix */
        if (myprocno == j%np) {
            col = (a+(j/np)*n);            /* top of jth col */
            akj = col + k;
            saxpy(n-k-1,-(*akj),work+1,1,akj+1,1);
            }
    }
}
```

A.2 PPOFA

```
#include <math.h>
#define ZERO (FLT)0.0e0
#define SQRT(x) (FLT)sqrt((FLT)(x))
extern int np, myprocno;
/*
 *   PPOFA - drives each node in a ring of p processors to
 *   factor a symmetric positive definite matrix, A, with
 *   type FLT(double) entries using the Cholesky algorithm.
 *   The matrix is distributed among the processors in a
 *   column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT -  pointer to vector containing columns
 *            of the matrix A to be factored. Only the lower
 *            triangle, including the diagonal is used.
 *     n      int - order of the matrix
 *   ON RETURN
 *     a      columns of L, the Cholesky triangle, in the
 *            lower triangle of the matrix A; the upper
 *            triangle of A remains unchanged. The
 *            factorization is not complete if info is not zero.
 *     info   *int - = 0 normal value
 *                   = k if the leading principle submatrix
 *                     of order k is found not to be positive
 *                     definite.
 *
 *   JULY 28,1989 - James W. Juszczak,
 *                  University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy, scopy
 *     amcom - broadcast, broad_rec
 */


ppofa(n,a,info)
FLT *a;
int n, *info;
```

```
{
int h, pk,                    /* pk: processor that owns col k */
    kcols;      /* # cols of kth submatrix held by processor */
register int k, i, j;
register FLT *cp, *wp, temp;
FLT *work, *col, *akk, *ajj;

*info = 0;
h = (myprocno < n%np) ? n/np+1: n/np;
work = a + h*n;
for (k = 0; k < n; k++) {
    wp = work;
    pk = k%np;
    kcols = h-((myprocno<pk)?k/np+1:k/np);
    if (myprocno == pk) {    /* processor that owns column k */
        col = a + (k/np)*n ;               /* top of kth col */
        akk = col + k;
        if ((temp = *akk) <= ZERO) {
            *work = (FLT)-1.0e0;
            *info = k+1;
            broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
            break;
            }
        *akk = SQRT(temp);
        temp = *akk;
        cp = akk + 1;
        *wp++ = *akk;
        for(i=k+1;i<n;i++)                      /* scale col k */
            *wp++ = *cp++/temp;
        broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
        scopy(n-k,work,1,akk,1);
        }
    else
        broad_rec(work);

    if (*work == (FLT)-1.0e0) {
        *info = k+1;
        return;
        }
    for (j = k+1; j < n; j++)            /* update submatrix */
        if (myprocno == j%np) {
            col = (a+(j/np)*n);            /* top of jth col */
            ajj = col + j;
            wp = work + (j-k);
            saxpy(n-j,-(*wp),wp,1,ajj,1);  /* update jth col */
            }
    }
}
```

A.3 PTRSL

```
#include <math.h>
#define ZERO (FLT)0.0e0
extern int np, myprocno;
/*
 *   PTRSL - drives each node in a ring of p processors
 *   to solve the upper triangular system of equations
 *   Ax=b by back substitution. The matrix is distributed
 *   among the processors in a column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT - is a pointer to a vector of type FLT
 *            which contains columns of the upper triangular
 *            matrix.
 *     n      int - order of the matrix
 *     t      *FLT - is a pointer to a vector of type FLT
 *            which contains the first (n-p+1) elements of
 *            b on  P(n) and 0 on all other processors.
 *     s      *FLT - is a pointer to a vector of type FLT
 *            which contains the last (p-1) elements of b
 *            on P(n) and 0 on all other processors.
 *   ON RETURN
 *     a      *FLT - is a pointer to a vector of type FLT
 *            which contains columns of the unchanged upper
 *            triangular matrix.
 *     x      *FLT - is a pointer to a vector of type FLT
 *            which contains the solution. The full solution
 *            vector is distributed among the processors such
 *            that x[j] is on the processor that owns column j
 *            of A, P(j).
 *     info   *int   = 0 normal value, A is non-singular
 *                   = k if the (k)th diagonal element of A
 *                     is zero
 *   NOVEMBER 2,1989 - James W. Juszczak,
 *                     University of Texas at Austin
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy
 *     amcom - send_left, rec_right
 */
```

```
ptrsl(n,a,x,t,s,info)
FLT *a,*x,*t,*s;
int n, *info;
{
int pj,                         /* processor that owns col j */
    tmp, size;
register int k, i, j;
register FLT xj, temp;
FLT *col, *ajj;

*info = 0;
size = np*sizeof(FLT);
k = (myprocno < n%np) ? n/np: n/np-1;
for (j = n-1; j >= 0; j--) {
    pj = j%np;
    if (myprocno == pj) {    /* processor that owns column j */
        col = a + (j/np)*n;              /* top of jth col */
        ajj = col + j;
        if (j != n-1) {
            rec_right(s);
            if (s[np-1] != 0.0) {  /* singularity detected ? */
                *info = (int)s[np-1];
                if (((myprocno != 0)?myprocno-1:np-1)!=(*info%np))
                    send_left(s,size);
                return;
                }
            }
        if ((xj = *ajj) != ZERO)
            temp = (j <= n-np) ? t[j] : 0.0;
        else {                       /* matrix is singular */
            *info = j;
            s[np-1] = (FLT)j;
            send_left(s,size);
            return;
            }
        xj = (s[0] + temp)/xj;
```

```
                                                          /* update s vector */
        if (j > 0) {
            for (i=0;i<np-2;i++) {
                if ((tmp = j-i-1) < 0) break;
                temp = (tmp < n-np+1) ? t[tmp] : 0.0;
                s[i] = s[i+1] - *(ajj-i-1) * xj + temp;
                }
            if (j >= (np-1))
                s[np-2] = - *(ajj-np+1)*xj + t[j-np+1];
            else
                s[np-2] = 0.0;
            }
        if (j != 0) send_left(s,size);      /* send s vector */
        saxpy(j-np+1,-xj,col,1,t,1);      /* update t vector */
        x[k--] = xj;
        }
    }
}
```

A.4 PQRDC

```
#include <math.h>
#define MIN(x,y) ((x) < (y) ? (x) : (y))
extern int np, myprocno;
/*
 *   PQRDC - drives each node in a ring of processors to form
 *   the orthogonal QR  decomposition of a general rectangular
 *   matrix, A, by applying Householder transformations.
 *   The matrix is distributed among the processors
 *   in a column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT -  pointer to a vector containing columns of
 *            the matrix A. Each column of A has been extended
 *            to a length of m+2 to store further information
 *            required to recover the Householder transformation
 *            at each step.
 *     m      int - number of rows in the matrix
 *     n      int - number of columns in the matrix
 *   ON RETURN
 *     a      *FLT -  pointer to a vector containing columns of
 *            the matrix A which has been overwritten by the
 *            triangular matrix R, on and above the diagonal.
 *            The vectors u which define the transformations,
 *            Uk = I -beta*uuT, are stored below the first
 *            subdiagonal. Beta and the first element of u are
 *            stored in A[m+1][k] and A[m+2][k] respectively.
 *            The orthogonal matrix Q = U1*U2* ... *Ur can be
 *            formed from these vectors.
 *     info   *int    = 0 normal value
 *                    = k if the (k)th column was already reduced
 *
 *   NOVEMBER 7,1989 - James W. Juszczak,
 *                     University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy, snrm2, sdot
 *     amcom - broadcast, broad_rec
 */
```

```
pqrdc(a,m,n,info)
FLT *a;
int m, n, *info;
{
register FLT beta, *col, *u, *work, n2, *akk;
FLT snrm2(), sdot();
register int j, k, r;

if (m<=0 || n<=0) return;
r = MIN(m-1,n);
*info = 0;
work = (FLT *) malloc((m+1)*sizeof(FLT));

for (k = 0; k < r; k++) {
               /* DETERMINE THE TRANSFORMATION, U=I-uuT*beta */
   if (myprocno == k%np) {
      *work = 0.0;
      col = a + (k/np)*(m+2);              /* top of kth col */
      akk = col+k;
      n2 = snrm2(m-k,akk,1);
      if (n2 == (FLT)0.0) {
         *work = (FLT)-1.0;
         *info = k;
         broadcast(work,(m-k+1)*sizeof(FLT),1,0,1,0);
         continue;
         }
      if (*akk < 0.0) n2 = -n2;
      *akk += n2;
      *(col+m) = beta = 1.0/(n2*(*akk));
      u = akk;
      broadcast(u,(m-k+1)*sizeof(FLT),1,0,1,0);
      *(col+m+1) = *u;
      }
   else {
      broad_rec(work);
      if (*work == (FLT)-1.0) {    /* rest of column is zero */
         *info = k;
         continue;
         }
      else {
         u = work;
         beta = *(u+m-k);
         }
      }

            /* PREMULTIPLY, A <- UA  or aj <- aj - u(uT)aj*beta */
   for (j = k+1; j < n; j++) { register FLT *akj, temp;
      if (myprocno == j%np) {
         col = (a+(j/np)*(m+2));              /* top of jth col */
         akj = col+k;
         temp = sdot(m-k,akj,1,u,1)*beta; /* temp = (uT)aj*beta */
         saxpy(m-k,-temp,u,1,akj,1);       /* aj <- aj - temp*u */
         }
      }
   if (myprocno == k%np) *akk = -n2;
   }
free(work);
}
```

A.5 PMGS

```
#include <math.h>
extern int np, myprocno;


/*
 *   PMGS - drives each node in a ring of processors to form
 *   an orthonormal basis which spans the column space of A,
 *   a general rectangular matrix with linearly independent
 *   columns. A is factored into the product QR, where Q has
 *   orthonormal columns and R is an upper triangular matrix
 *   with positive diagonal elements. A is overwritten by Q.
 *   The matrix is distributed among the processors in a column
 *   wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT -  pointer to a vector containing columns
 *            of the matrix A.
 *     m      int - number of rows in the matrix
 *     n      int - number of columns in the matrix
 *   ON RETURN
 *     a      *FLT -  pointer to a vector containing columns of
 *            the matrix A which has been overwritten by Q, a
 *            matrix with orthonormal columns.
 *     r      *FLT -  pointer to vector containing the n x n upper
 *            triangular matrix R distributed to the processors in
 *            column wrapped fashion.
 *     info   *int   = 0 normal value
 *                   = k if the (k)th column was linearly dependent
 *
 *   NOVEMBER 12,1989 - James W. Juszczak,
 *                      University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - sscal, saxpy, snrm2, sdot
 *     amcom - broadcast, broad_rec
 */
```

```
pmgs(m,n,a,r,info)
FLT *a, *r;
register int m, n;
int *info;
{
register FLT *aj, *ak, *qk, temp, *rkj, *work;
FLT snrm2(), sdot();
register int j, k;

if (m<=0 || n<=0) return;
*info = 0;
work = (FLT *) malloc(m*sizeof(FLT));
ak = a;                          /* points to top of kth col of A */

for (k = 0; k < n; k++) {
   if (myprocno == k%np) {
      *work = (FLT)0.0;
      *(r+k) = temp = snrm2(m,ak,1);
      if (temp == (FLT)0.0) {
         *work = (FLT)-1.0;
         broadcast(work,sizeof(FLT),1,0,1,0);
         *info = k;
         return;
         }
      sscal(m,(FLT)1.0/temp,ak,1);
      qk = ak;
      broadcast(qk,m*sizeof(FLT),1,0,1,0);
      r += n;
      ak += m;
      }
   else {
      broad_rec(work);
      if (*work == (FLT)-1.0) {
         *info = k;
         return;
         }
      else
         qk = work;
      }
   rkj = r+k;
   for (j = k+1; j < n; j++) {
      if (myprocno == j%np) {
         aj = a + (j/np)*m;
         *rkj = temp = sdot(m,aj,1,qk,1);
         saxpy(m,-temp,qk,1,aj,1);
         rkj += n;
         }
      }
   }
free(work);
}
```

A.6 PGEHR

```
#include <math.h>
extern int np, myprocno;

/*
 *   PGEHR - drives each node in a ring of p processors to
 *   reduce a square general matrix, A, to upper Hessenberg
 *   form by determining and applying Householder unitary
 *   similarity transformations. The matrix is distributed
 *   among the processors in a column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT -  pointer to vector containing columns of
 *            the matrix A. Each column of A has been extended
 *            to a length of n+2 to store further information
 *            required to recover the Householder transformation
 *            at each step.
 *     n      int - order of the matrix
 *   ON RETURN
 *     a      *FLT -  pointer to vector containing
 *            columns of the matrix A which has been overwritten
 *            by the upper Hessenberg matrix above the second
 *            sub-diagonal. The vectors u which define the
 *            transformations, Uk = I -beta*uuT, are stored
 *            below the first subdiagonal. Beta and the first
 *            element of u are stored in A[n+1][k] and A[n+2][k]
 *            respectively.
 *     info   *int   = 0 normal value
 *                   = k if the (k)th column was already reduced
 *
 *   NOVEMBER 2,1989 - James W. Juszczak,
 *                     University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy, snrm2, sdot, scopy
 *     amcom - broadcast, broad_rec, dvec_add, exchange
 */
```

```
pgehr(a,n,info)
FLT *a;
int n, *info;
{
register FLT beta, *col, *u, *work, *x, *y, n2, *ak1k;
FLT snrm2(), sdot(), ZERO[1];
register int j, k, tmp;

*ZERO=(FLT)0.0;
*info = 0;
work = (FLT *) malloc((3*n+np)*sizeof(FLT));
x = work + n;
y = x + n;
for (k = 0; k < n-2; k++) {
                        /* DETERMINE THE TRANSFORMATION, U=I-uuT*beta */
   if (myprocno == k%np) {
   register FLT temp;
      col = a + (k/np)*(n+2);                    /* top of kth col */
      ak1k = col+k+1;
      n2 = snrm2(n-k-1,ak1k,1);
      if (n2 == (FLT)0.0) {
         *work = (FLT)-1.0;
         *info = k;
         broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
         continue;
         }
      if (*ak1k < (FLT)0.0) n2 = -n2;
      *ak1k += n2;
      *(col+n) = beta = 1.0/(n2*(*ak1k));
      u = ak1k;
      broadcast(u,(n-k)*sizeof(FLT),1,0,1,0);
      *(col+n+1) = *u;
      }
   else {
      broad_rec(work);
      if (*work == (FLT)-1.0) {          /* rest of column is zero */
         *info = k;
         continue;
         }
```

```
     else {
        u = work;
        beta = (*(u+n-k-1));
        }
     }
            /* PREMULTIPLY, A <- UA  or aj <- aj - u(uT)aj*beta */
  for (j = k+1; j < n; j++) { register FLT *ak1j, temp;
      if (myprocno == j%np) {
        col = (a+(j/np)*(n+2));                   /* top of jth col */
        ak1j = col+k+1;
        temp = sdot(n-k-1,ak1j,1,u,1)*beta;/* temp = (uT)aj*beta */
        saxpy(n-k-1,-temp,u,1,ak1j,1);      /* aj <- aj - temp*u */
        }
     }
          /* POSTMULTIPLY, A <- AU or aj <- aj - aj*u(ut)*beta) */
  scopy(n,ZERO,0,x,1);
  for (j = k+1; j < n; j++) { register FLT ujk;
     if (myprocno == j%np) {
        col = (a+(j/np)*(n+2));                   /* top of jth col */
        ujk = *(u+j-k-1);
        saxpy(n,ujk,col,1,x,1); /* x<-x+aj*u[j] for each aj on P */
        }
     }
  scopy(n,ZERO,0,y,1);                     /* y <- sum x over all P's */
  tmp = n/np;
  if (n%np != 0) tmp++;
  dvec_add(x,n,tmp*myprocno+y);
  exchange(y,tmp*sizeof(FLT));
                                        /* complete submatrix update  */
  for (j = k+1; j < n; j++) { register FLT temp;
     if (myprocno == j%np) {
        col = (a+(j/np)*(n+2));                   /* top of jth col */
        temp = *(u+j-k-1) * beta;
        saxpy(n,-temp,y,1,col,1);           /* aj <- aj - u[j]*y */
        }
     }
  if (myprocno == k%np) *ak1k = -n2;
  }                                       /* end outermost FOR loop */
free(work);
}
```

BIBLIOGRAPHY

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

[2] Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins Press, 1983

[3] Ipsen, I.C.F., Saad, Y., and Schultz, M. H., "Complexity of Dense-Linear-System Solution on a Multiprocessor Ring," *Linear Algebra and its Applications*, 77:205-239, 1986

[4] Juszczak, J.W. and van de Geijn, R.A., "An Experiment in Coding Portable Parallel Matrix Algorithms," *Proceedings of the Fourth Annual Hypercube Conference*, to appear, 1989

[5] Li, G. and Coleman, T.F., "A Parallel Triangular Solver for a Distributed-Memory Multiprocessor," *Siam J. Sci. Stat. Comput.*, Vol. 9, No. 3, May 1988

[6] Li, G. and Coleman, T.F., "A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors," *Siam J. Sci. Stat. Comput.*, Vol. 10, No. 2, March 1989

[7] McBryan, O.A. and Van de Velde, E.F., "Hypercube Algorithms and Implementations," *SIAM J. Sci. Stat. Comp.*, Vol. 8, No. 2, March 1987

[8] O'Leary, D.P. and Stewart, G.W., "Data-Flow Algorithms for Parallel Matrix Computations," *Communications of the ACM*, Vol. 28, No. 8, August 1985

[9] Saad, Y. and Schultz, M.H., "Data Communication in Parallel Architectures," Yale University, Research Report YALEU/DCS/RR-461, 1986

[10] Seitz, C.L., Seizovic, J., and Su, W., "The C Programmer's Abbreviated Guide to Multiprocessor Programming," Caltech Computer Science Technical Report Caltech-CS-TR-88-1, 1988.

[11] Stewart, G.W., *Introduction to Matrix Computations*, Academic Press, 1973

[12] Van de Geijn, R. A., "A Novel Storage Scheme for Parallel Jacobi Methods," The University of Texas at Austin, Dept. of Computer Sciences, TR-88-26

[13] Van de Geijn, R. A., "Storage Schemes for Parallel Eigenvalue Algorithms," to appear in *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms,* G. Golub and P. Van Dooren (Ed.), NATO ASI Series, Springer Verlag, 1989

[14] Van de Geijn, R. A., "Machine Independent Parallel Numerical Algorithms," in *Parallel Supercomputing: Methods, Algorithms and Applications*, Graham F. Carey (Ed.), Wiley, 1989

[15] Van de Geijn, R. A., "Performance Evaluation of the AMETEK 2010: A Preliminary Report," unpublished manuscript

[16] Van de Geijn, R. A. and D.G. Hudson III, "Efficient Parallel Implementation of the Nonsymmetric QR Algorithm," in *Proceedings of the Fourth Annual Hypercube Conference*, to appear, 1989