A PARALLEL IMPLEMENTATION OF A SIMULTANEOUS ITERATION ALGORITHM FOR CALCULATION OF NESTED INVARIANT SUBSPACES OF LARGE NON-HERMITIAN MATRICES

by

John Robert Meyer

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1991

Advisory Committee:
Professor G. W. Stewart, Chairman/Advisor
Professor Diane P. O'Leary
Associate Professor Howard Elman
Associate Professor Clyde Kruskal

PREFACE

This thesis implements a parallel implementation of a simultaneous iteration technique for computing a nested sequence of orthonormal bases for the dominant invariant subspaces of a non-Hermitian matrix. This method is particularly suited to the calculation of several eigenvalues and eigenvectors of large sparse matrices since the only requirement is that one compute the product of the matrix with a vector, a computation that can be performed very quickly on a parallel computer architecture. Convergence of the method is improved through the application of a Schur-Rayleigh-Ritz step. Since this iterative method can cause loss of orthogonality in the basis vectors of the invariant subspaces, a QR reorthogonalization step is also done in parallel and the results of the executions are examined.

ACKNOWLEDGEMENTS

My sincere thanks to Peter Whitman, Neil Vanderlipp, Fred Nelker, and the group at the University of Maryland Institute for Advanced Computer Studies, without whose help this thesis would not have been possible; and my appreciation to Dr. G. W. Stewart for his patience during the long completion of this thesis.

TABLE OF CONTENTS

Section	Page
List of Figures	v
Chapter 1. The Theory of Invariant Subspaces	1
Chapter 2. The Method of Simultaneous Iteration	4
Chapter 3. The Overall Algorithm and the DOMINO Environment	8
Chapter 4. Parallel Calculation of AQ_v	13
Chapter 5. Parallel Reorthogonalization	18
Chapter 6. Parallel Reorthogonalization with Binary Summation	22
Chapter 7. Empirical Calculation of ϕ, σ, and τ	31
Chapter 8. Results of the Parallel AQ Algorithm	36
Chapter 9. Results of the Parallel Reorthogonalization Algorithm	43
Chapter 10. Conclusions	47
Appendix 1. Table of Symbols	49
Appendix 2. C Source Code for the Implementation	54
Appendix 3. Related Methods	101
References	103

LIST OF FIGURES

Figure	Description	Page
1.	Basic Algorithm for This Implementation	9
2.	Conformal Partitioning of Blocks	13
3.	Algorithm for the General Parallel Computation of AQ_v	14
4.	Algorithm for Parallel Reorthogonalization	19
5.	Binary Summation Example for 8 Processors	22
6.	Algorithm for Calculation of Σ_i when $p = 2^k$	25
7.	Algorithm for Calculation of Π_i when $p = 2^k$	26
8.	Binary Summation Example for 7 Processors	27
9.	Algorithm for Calculation of the Endpoints of a Partition	28
10.	Algorithm for the General Calculation of Σ_i	29
11.	Algorithm for the General Calculation of Π_i	29
12.	Algorithm for the Empirical Calculation of ϕ	31
13.	Empirical Calculation Loop for σ_b	32
14.	Empirical Calculation Loop for σ_s	33

Figure	Description	Page

15.	Times Used in the Determination of σ_b, σ_s, and τ	34
16.	Results of Random Diagonal Matrix Example with $n = 100$, $m = 4$, and $\phi = 882\mu$secs	37
17.	Total Costs for Random Diagonal Matrix Example	38
18.	A 5×5 Triangular Grid	39
19.	Mapping of the Points of the Grid to q_j for $\gamma = 5$	40
20.	Results of Markov Chain Matrix Calculation with $\gamma = 10, n = 55$, and $m = 4$	42
21.	Results of the Summation-Independent Calculations	43
22.	Transmission Costs for the Parallel Reorthogonalization	44
23.	Total Costs for the Parallel Reorthogonalization Algorithm	45
24.	Results of the Linear and Binary Summations	46

CHAPTER 1

The Theory of Invariant Subspaces

This chapter presents an overview of the theory of invariant subspaces and shows how they provide a partial solution to the eigenvalue problem. For more details see Stewart [11, 12].

An *invariant subspace* of a matrix $A \in \mathbb{C}^{n \times n}$ is any subspace $\mathbf{\Gamma} \subset \mathbb{C}^n$ with the property that

$$x \in \mathbf{\Gamma} \quad \Rightarrow \quad Ax \in \mathbf{\Gamma} \,. \tag{1.1}$$

Let the set of eigenvalues of A be denoted by $\Lambda_A \overset{\text{def}}{=} \{\lambda_i\}_{i=1}^n$, and without loss of generality assume that the eigenvalues are indexed so that

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

If $\mathbf{\Gamma}$ is an invariant subspace of A and the columns of a matrix $Q \overset{\text{def}}{=} [q_1,\ q_2,\ \cdots,\ q_m] \in \mathbb{C}^{n \times m}$, $m \leq n$, form an orthonormal basis for $\mathbf{\Gamma}$ then $Aq_j \in \mathbf{\Gamma}$ and can be expressed as a linear combination of the q_i:

$$Aq_j = \sum_{i=1}^{m} \tau_{ij} q_i \,, \quad 1 \leq j \leq n \,.$$

Now let $t_j \overset{\text{def}}{=} \left[\tau_{1j},\ \tau_{2j},\ \cdots,\ \tau_{mj}\right]^{\mathrm{T}} \in \mathbb{C}^m$ and let $T \overset{\text{def}}{=} [t_1,\ t_2,\ \cdots,\ t_m] \in \mathbb{C}^{m \times m}$, so that $Aq_j = Qt_j$, $1 \leq j \leq n$, and

$$AQ = QT \,. \tag{1.2}$$

If $\hat{x} \in \mathbb{C}^m$ is an eigenvector of T corresponding to eigenvalue $\hat{\lambda} \in \Lambda_T$, then by (1.2) and the fact that $T\hat{x} = \hat{\lambda}\hat{x}$, then it follows that $Q\hat{x}$ is an eigenvector of A corresponding to $\hat{\lambda}$ and [12]

$$\Lambda_T \subset \Lambda_A . \tag{1.3}$$

Conversely, let $\hat{\lambda} \in \Lambda_A$ with associated eigenvector $\hat{x}$. Since the columns of Q form an orthonormal basis for $\mathbf{\Gamma}$, there exists a vector $\hat{y} \in \mathbb{C}^m$ such that $\hat{x} = Q\hat{y}$. Then we have $A\hat{x} = \hat{\lambda}\hat{x} \Rightarrow AQ\hat{y} = \hat{\lambda}Q\hat{y} \Rightarrow Q^{\mathrm{H}}AQ\hat{y} = \hat{\lambda}\hat{y} \Rightarrow T\hat{y} = \hat{\lambda}\hat{y}$ so that $\hat{\lambda} \in \Lambda_T$. Thus the knowledge of an invariant subspace of A allows us to find the eigenvectors in that subspace, along with their associated eigenvalues, by finding the eigenvectors of the smaller matrix T.

The program developed in this thesis attempts to calculate a nested sequence of orthonormal bases for the dominant invariant subspaces of A. The existence of such a basis is guaranteed by the well-known Schur decomposition, which states that if $A \in \mathbb{C}^{n \times n}$ then there exists a unitary matrix $Q \in \mathbb{C}^{n \times n}$ such that the matrix $Q^{\mathrm{H}}AQ$ is upper triangular with the eigenvalues of A appearing in descending order of absolute magnitude along its diagonal [3]. Since A is not in general Hermitian, the columns of Q are not eigenvectors, although they enjoy many of the properties of eigenvectors. If

$$|\lambda_{r-1}| > |\lambda_r| > |\lambda_{r+1}|$$

$1 \le r \le n$, then q_r is uniquely determined up to a factor of ± 1 [11]. If

$$|\lambda_{r-1}| > |\lambda_r| = |\lambda_{r+1}| = \cdots = |\lambda_s| > |\lambda_{s+1}|,$$

$1 \le r < s \le n$, then the vectors $\{q_j\}_{j=r}^{s}$ are not uniquely determined; however, the subspace spanned by the vectors is unique [11]. The columns of Q are called *Schur vectors* of A.

If $|\lambda_r| > |\lambda_{r+1}|$, $1 \le r < n$, then the vectors $\{q_j\}_{j=1}^{r}$ form an orthonormal basis for the unique *dominant invariant subspace* $\mathbf{\Gamma}_r$ corresponding to $\{\lambda_i\}_{i=1}^{r}$ [11]. To see this, let $Q^{|r} \stackrel{\text{def}}{=} [q_1, q_2, \cdots, q_r] \in \mathbb{C}^{n \times r}$ and let $T^{\overline{|r}} \in \mathbb{C}^{r \times r}$ be a leading principal submatrix of T. From (1.2) and the upper triangularity of T,

$$AQ^{|r} = Q^{|r}T^{\overline{|r}} \tag{1.4}$$

implying that the columns of $Q^{|r}$ span an invariant subspace $\mathbf{\Gamma}_r$ of A whose eigenvalues are $\Lambda_{T^{\overline{|r}}}$ and if $T^{\overline{|r}}\hat{x}_i = \hat{\lambda}_i\hat{x}_i$ then

$$AQ^{|r}\hat{x}_i = Q^{|r}T^{\overline{|r}}\hat{x}_i = \hat{\lambda}_i Q^{|r}\hat{x}_i \ ,$$

so that $Q^{|r}\hat{x}_i$ is an eigenvector of A corresponding to eigenvalue $\hat{\lambda}_i$ [11].

In order to avoid the expense of complex arithmetic, we will use a variant of the Schur form for real matrices in which T is block upper triangular with 1×1 and 2×2 blocks along the diagonal. The existence of such a block decomposition is established in [3].

CHAPTER 2

The Method of Simultaneous Iteration

The dominant Schur vectors of A will be calculated by *simultaneous iteration*, a generalization of the power method [3, 13]. Recall that the power method calculates the dominant eigenvector of a matrix by generating a sequence of approximating vectors according to the iteration $q_{\nu+1} = Aq_\nu \rho_{\nu+1}^{-1}$ for $\nu = 0, 1, 2, \cdots,$ where $\rho_{\nu+1}$ is a nonzero scaling factor. The method is particularly attractive when A is large and sparse since it requires only that the user be able to multiply a vector by A [11], an operation that can be coded to exploit any *a priori* knowledge of A. Simultaneous iteration enjoys the same freedom of implementation, and its simplicity lends itself nicely to a parallel computer architecture. Computational properties of this method are discussed extensively by Stewart [11] and a synopsis of these properties is given here.

The method of simultaneous iteration had as its prototype the *Treppeniteration* ("staircase iteration") first proposed by Bauer [1]. The idea is to begin with a matrix $Q_0 \in \mathbb{C}^{n \times m}$ whose columns form a basis for some m-dimensional subspace $\boldsymbol{\Psi}$ and generate a sequence of matrices according to

$$AQ_\nu = Q_{\nu+1}R_{\nu+1}, \quad \nu = 0, 1, 2, \cdots \tag{2.1}$$

where $R_{\nu+1}$is a nonsingular matrix. Since the column spaces of Q_ν and $Q_\nu R_\nu$ are the same, R_ν can be considered a scaling factor [11]. It is easy to demonstrate that repeated application of (2.1) gives

$$A^\nu Q_0 = Q_\nu R_\nu R_{\nu-1} \cdots R_1$$

so that the columns of Q_ν form a basis for $A^\nu \mathbf{\Psi}$ which approaches $\mathbf{\Gamma}$, the invariant subspace of A [11]. Specifically, under mild restrictions on Q_0, if $|\lambda_r| > |\lambda_{r+1}|$ the column space of Q_ν approaches $\mathbf{\Gamma}_r$ [11].

In most steps the matrix $R_{\nu+1}$ can be taken to be the $m \times m$ identity matrix so that (2.1) becomes

$$Q_{\nu+1} = AQ_\nu, \quad \nu = 0, 1, 2, \cdots \tag{2.2}$$

However, since A is not necessarily unitary, the columns of $Q_{\nu+1}$ may approach dependency. To guard against this, $R_{\nu+1}$ can be used to scale the matrix AQ_ν so that its columns are once again orthogonal. This reorthogonalization can be done by applying Householder transformations to the columns of AQ_ν; however, we will use another algorithm which for our purposes is just as stable and parallelizes simply. It is easy to show informally that if δ is the maximum number of decimal digits that can be allowed to be lost between orthogonalizations and $\kappa(T) \stackrel{\text{def}}{=} \|T\|\|T^{-1}\|$ then

$$\eta \stackrel{\text{def}}{=} \left\lfloor \frac{\delta}{\log_{10} \kappa(T)} \right\rfloor \tag{2.3}$$

is the number of iterations that can be performed safely with $R_{\nu+1} = I_m$ [11].

The norm $\|\cdot\|$ is the Frobenius norm given by $\|A\|_{\mathrm{F}} \stackrel{\text{def}}{=} \sqrt{\text{trace}\,(A^{\mathrm{H}}A)} = \left[\sum_{i=1}^{n}\sum_{j=1}^{n}\bar{\alpha}_{ji}\alpha_{ij}\right]^{1/2}$. Since T is upper triangular, the calculation of $\kappa(T)$ is relatively inexpensive.

The rate of convergence of the rth column of Q_ν is proportional to

$$\max\left\{\left|\frac{\lambda_r}{\lambda_{r-1}}\right|^{\nu}, \left|\frac{\lambda_{r+1}}{\lambda_r}\right|^{\nu}\right\}$$

and if (nearly) equimodular eigenvalues exist, this convergence may be intolerably slow [12]. Stewart [11, 12] describes a method, which he calls a *Schur-Rayleigh-Ritz* (or *SRR*) *step*, whereby the process may be accelerated. A matrix $B_\nu \in \mathbb{C}^{m\times m}$ is formed according to

$$B_\nu = Q_\nu{}^{\mathrm{H}} A Q_\nu$$

and reduced to Schur form $T_\nu \in \mathbb{C}^{m\times m}$ by a unitary similarity transformation

$$T_\nu = Y_\nu{}^{\mathrm{H}} B_\nu Y_\nu. \tag{2.4}$$

Then Q_ν is overwritten

$$Q_\nu \leftarrow Q_\nu Y_\nu. \tag{2.5}$$

In general, if $|\lambda_{i-1}| > |\lambda_i| > |\lambda_{i+1}|$ then the ith column of Q_ν, $q_i{}^{(\nu)}$, will converge at a rate proportional to [12]

$$\left|\frac{\lambda_{m+1}}{\lambda_i}\right|^{\nu}.$$

Therefore, convergence is accelerated and the first columns of Q_ν tend to

converge faster than the later ones [12]. This latter fact saves work in the iteration since once they have converged, the left-most columns need not participate in the basic iteration; only the periodic reorthogonalizations and SRR steps must include them.

If $|\lambda_{i-1}| \approx |\lambda_i|$ or $|\lambda_i| \approx |\lambda_{i+1}|$, the ith column cannot be computed accurately and a convergence criterion based on the $q_i^{(\nu)}$ settling down is likely to fail, even though the subspace they span may converge [12]. Thus, let $t_i^{(\nu)} \in \mathbb{C}^m$ be the ith column of T_ν in (2.4) and let $r_i^{(\nu)} \stackrel{\text{def}}{=} Aq_i^{(\nu)} - Q_\nu t_i^{(\nu)} \in \mathbb{C}^m$ be the ith residual vector. The ith column of Q_ν produced in (2.5) is said to have converged if $\|r_i^{(\nu)}\|_2$ is less than some error tolerance [12]. If this condition is met for all of the $q_i^{(\nu)}, 1 \leq i \leq m$, then the residual matrix $R_\nu \stackrel{\text{def}}{=} [r_1, r_2, \cdots, r_m] \in \mathbb{C}^{n \times m}$ is small [12]. Although this condition cannot guarantee the accuracy of the columns of Q, it does imply that there is a small matrix $E_\nu \stackrel{\text{def}}{=} -R_\nu Q_\nu^{\mathrm{H}} \in \mathbb{C}^{n \times n}$ such that

$$(A + E_\nu)Q_\nu = Q_\nu T_\nu,$$

so that Q_ν and T_ν are the matrices associated with the slightly perturbed matrix $A + E_\nu$ [12]. Nearly equimodular eigenvalues should be grouped together to avoid inadvertently including some small eigenvalue of $A + E_\nu$ in T_ν and their residual vectors tested only after the average value of the eigenvalues has converged [12].

CHAPTER 3

The Basic Algorithm and the DOMINO Environment

The program implemented is a C language translation of the FORTRAN program SRRIT written by Stewart [12]. The parallel sections replace the sequential calculation of AQ_ν and the Gram-Schmidt reorthogonalization step. This chapter will concentrate on the parallel procedures, but first we will give an outline and discussion of the basic algorithm. Pseudocode for the algorithm is given in Figure 1 and the parallel sections are shown in boxes.

Program initialization consists of several parts. First, the iteration counter, *Iteration*, is set to 0. *LeftColumn* is the index of the first column of Q that has not yet converged an initially points to the first column. The matrix Q may be supplied by the user or set to have random elements by the program. The columns of Q may be automatically orthogonalized if desired.

The *SRR* loop is the main loop of the program. Each time it is executed, an SRR step is performed, setting the new value of Q, AQ (which holds the product $A \times Q$), and $T = Q^{\mathrm{T}}AQ$. Next, the residuals are calculated and convergence is tested. As mentioned in Chapter 2, nearly equimodular eigenvalues of T are grouped together by procedure **group** as are the eigenvalues from the previous step. Furthermore, the root-mean square average of the

```
            Iteration ← 0;
            LeftColumn ← 1;
            initialize Q;
            input the number of vectors to calculate, NumberOfVectors;
SRR:        loop
                perform an SRR step, calculating AQ = A × Q and T;
                compute the residuals;
                check convergence, resetting LeftColumn if necessary;
                exit SRR when LeftColumn > NumberOfVectors or too many iterations;
                compute NextSRR, OrtInterval, and NextOrt;
                Q ← AQ;
                Iteration ← Iteration + 1;
ORTH:           while Iteration < NextSRR loop
POWER:              while Iteration < NextOrt loop
                        AQ ← A × Q;
                        Q ← AQ;
                        Iteration ← Iteration + 1;
                    end loop POWER;
                    orthogonalize Q;
                    NextOrt ← min {NextSRR, Iteration + OrtInterval};
                end loop ORTH;
            end loop SRR;
            NumberOfVectors ← LeftColumn − 1;
```

Figure 1.

Basic Algorithm for This Implementation.

norms of the residuals is computed and returned by **group**. If the two groups have the same number of eigenvalues and the average value of the eigenvalues has stabilized, then the residuals are averaged and tested against an error tolerance. If the test succeeds, then *LeftColumn* is incremented by the number of eigenvalues in the group and the tests are repeated for the next group. Otherwise, the two termination conditions are tested and control exits the *SRR* loop if either is met.

The iteration at which the next SRR step is taken (*NextSRR*), the inter-

val between orthogonalizations (*OrtInterval*), and the iteration at which the next reorthogonalization will take place (*NextOrt*) are then calculated from the RMS average returned by **group** and the value of η in (2.3). Finally, the SRR iteration counter is incremented and the product $A \times Q$ calculated at the beginning of the current iteration is assigned to Q.

The *ORTH* loop performs the basic iteration as well as reorthogonalization and begins by executing the *POWER* loop which repeatedly applies (2.2) until a reorthogonalization step is required. When the iteration counter reaches the value of *NextOrt*, Q is reorthogonalized and *NextOrt* is set to the iteration at which the next SRR step is to be performed or the iteration at which the next reorthogonalization step is to be performed, whichever comes first.

This completes the *ORTH* and *SRR* loops. Finally, *NumberOfVectors* is set to the number of vectors that have converged.

The parallel sections are coded using the DOMINO parallel message passing environment developed at the University of Maryland for medium-grained parallel computation [6]. Computations in the DOMINO environment take place in computational *nodes* that compute and exchange data with other nodes that can be either on the same physical processor or on different ones. Each node has an associated *node program* responsible for performing the node's calculations; although each node has only one node program, each

node program may serve several nodes. Data areas for each node are allocated by the DOMINO system when the node is created and initialized data may be passed to each node through a structure called *auxiliary storage*. Note that under the DOMINO environment, a single processor may (and usually does) host many nodes.

The user is responsible for supplying two important nodes for each processor: a ‘‘boot node’’ and a ‘‘go node.’’ The boot node program must be a C function called **boot** and is responsible for initializing the DOMINO system’s storage pools and for creating the go node. This use of a boot node allows a computational network to create itself according to each application and as DOMINO programs are ported from one computer to another, only the boot node program requires recoding to exploit the underlying hardware. Once the boot node has finished, execution begins in the go node just created. This node program must be a C function called **go** and it creates the other nodes to be used on the processor.

From this point, the executions of the nodes on the processor are scheduled in some way, typically in a round-robin fashion. DOMINO is not an interrupt-driven environment in which each node gets a slice of time before it loses control of the processor; each node must in good faith relinquish control when necessary, typically while waiting for data from another node. Incoming messages for each node are queued as they arrive, and a node that

has suspended itself is guaranteed not to be reawakened until all of its data requests have been satisfied. This scheme gives rise to an important feature of the DOMINO system called *determinacy*. Informally, determinacy is the property that guarantees that the results of a DOMINO program are independent of how its computational nodes are assigned to processors and how the nodes are scheduled on the processors, allowing DOMINO programs to be debugged on single-processor machines (where program development tools are often more sophisticated) and moved to true parallel machines with little or no modification [6].

The program written for this thesis uses a ring architecture of p processors as the computational network. Each processor hosts, in addition to the required boot node and go node, a node for calculating the product AQ_ν (called an *AQ node*) and a node for computing the reorthogonalization (called a *QR node*). One processor in the ring is designated a *master* and is responsible for interfacing the nodes of the ring with a *control* node on the same processor. This control node is a go node that performs the larger sequential tasks of Figure 1 and activates the ring when necessary. The AQ and QR nodes of the ring are numbered from 0 to $p - 1$ with node 0 indexing the master node. Since each physical processor used in this thesis has only one of each type of node, we will assume that AQ node i and QR node i reside on processor i.

CHAPTER 4

Parallel Calculation of AQ_ν

The parallel calculation of AQ_ν is performed in the following way. If explicit storage for the matrix A is needed, it may be divided into p rows of p blocks, each of which can then be placed in the auxiliary data storage area of each AQ node as part of program initialization. In many instances, however, no explicit representation of A is required, as will be shown later. In any case, Q_ν is treated as a vector of p blocks, each dimensioned conformally with its corresponding block of A, roughly n/p:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & \cdots & A_{0,p-1} \\ A_{10} & A_{11} & A_{12} & \cdots & A_{1,p-1} \\ A_{20} & A_{21} & A_{22} & \cdots & A_{2,p-1} \\ & & \vdots & & \\ A_{p-1,0} & A_{p-1,1} & A_{p-1,2} & \cdots & A_{p-1,p-1} \end{bmatrix} \begin{bmatrix} Q_0{}^{(\nu)} \\ Q_1{}^{(\nu)} \\ Q_2{}^{(\nu)} \\ \vdots \\ Q_{p-1}^{(\nu)} \end{bmatrix}$$

Figure 2.

Conformal Partitioning of Blocks.

Note that the blocks in Figure 2 are indexed beginning with 0, a departure from the normal indexing conventions of mathematics. Indexing in this way, however, will facilitate our discussion of the parallel algorithms and should come naturally to most computer scientists.

```
DISTRIBUTE:   for k in p − i − 1 downto 1 loop
                  if node i is the master then
                      read Q_k^(v) from memory;
                      send Q_k^(v) to clockwise neighbor;
                  else
                      receive Q_k^(v) from counter-clockwise neighbor;
                      send Q_k^(v) to clockwise neighbor;
                  end if;
              end loop DISTRIBUTE;
              if node i is the master then
                  read Q_i^(v) from memory;
              else
                  receive Q_i^(v) from counter-clockwise neighbor;
              end if;

              Q_k^(v+1) ← 0;
SUM:          for k in i wrap around p − 1 to i − 1 loop
                  Q_i^(v+1) ← Q_i^(v+1) + A_ik × Q_k^(v);
                  send Q_k^(v) to clockwise neighbor;
                  receive next Q_k^(v) from counter-clockwise neighbor;
              end loop SUM;

COLLECT:      if node i is the master then
                  place Q_i^(v+1) in memory;
              else
                  send Q_i^(v+1) to counter-clockwise neighbor;
              end if;
              for k in 1 to p − i − 1 loop
                  receive Q_k^(v+1) from clockwise neighbor;
                  if node i is the master then
                      place Q_k^(v+1) in memory;
                  else
                      send Q_k^(v+1) to counter-clockwise neighbor;
                  end if;
              end loop;
```

Figure 3.

Algorithm for the General Parallel Computation of AQ_v.

Figure 3 presents an algorithm for the parallel calculation of AQ_ν for node i. The blocks of Q_ν are distributed clockwise around the ring of processors at each multiplication step. The blocks are circulated around the ring in the *SUM* loop so that each node forms the inner product of its row of blocks with Q_ν, producing the ith block of $Q_{\nu+1}$, ${Q_i}^{(\nu+1)}$. Here the pseudocode construction "**for** k **in** i **wrap around** $p-1$ **to** $i-1$ **loop**" means that the loop index k takes on the values $i,\ i+1,\ i+2,\ \cdots,\ p-1$ and then wraps around to $0,\ 1,\ 2,\ \cdots,\ i-1$. Node i then sends its completed block counter-clockwise followed by those of its clockwise neighbors.

The analysis of the parallel algorithm can be divided into two separate analyses of calculation time and communication time. We will use three functions defined as

$$T_{\text{cal}} = T_{\text{cal}}(p, n, m;\ \phi)$$

$$T_{\text{com}} = T_{\text{com}}(p, n, m;\ \sigma, \tau)$$

$$T_{\text{tot}} = T_{\text{tot}}(p, n, m;\ \phi, \sigma, \tau) \stackrel{\text{def}}{=} T_{\text{cal}} + T_{\text{com}}$$

As is customary in this type of analysis, let ϕ be the time necessary to execute a floating point multiplication and addition, along with any indexing and looping overhead involved. For the communication time, assume that a fixed startup time σ is required for the transmission and reception of a message between nodes, regardless of the length of the message, and that data can then be transferred at a rate of τ^{-1} items per unit time. Let the startup time

required for passing data in a bucket brigade fashion to be σ_b and let σ_s be the startup time required when all nodes are simultaneously transferring data (see Chapter 7). Then the three steps in the calculation of AQ_ν are as follows:

1. Distribute the blocks of Q_ν around the ring of processors with cost

$$T_1 = (p-1)\left(\sigma_\mathrm{b} + \frac{mn}{p}\tau\right) \tag{4.1}$$

2. Calculate $A_{ik}{Q_k}^{(\nu)}$ with transmission cost

$$T_2 = p\left(\sigma_\mathrm{s} + \frac{mn}{p}\tau\right) \tag{4.2}$$

and calculation cost $\frac{mn^2}{p^2}\phi$.

3. Collect the block sums from the ring of processors with cost

$$T_3 = (p-1)\left(\sigma_\mathrm{b} + \frac{mn}{p}\tau\right) \tag{4.3}$$

Summing the calculation and transmission times in the three steps gives

$$T_\mathrm{cal} = \frac{mn^2\phi}{p} \tag{4.4}$$

$$T_\mathrm{com} = p\sigma_\mathrm{s} + mn\tau + 2(p-1)\left(\sigma_\mathrm{b} + \frac{mn}{p}\tau\right). \tag{4.5}$$

The quantity T_cal decreases as more processors are added. Eventually, however, T_com will dominate; that is, our ability to speed up the algorithm by adding processors is limited by the communication overhead.

Let us find the number of processors that can be used profitably. We

find the minimum point of the function T_{tot} by differentiating with respect to p

$$\frac{\partial T_{\text{tot}}}{\partial p} = \frac{2mn\tau - mn^2\phi}{p^2} + \sigma_{\text{s}} + 2\sigma_{\text{b}},$$

setting this equation to 0, and solving for p_{opt}, finding

$$p_{\text{opt}} = \left[\frac{mn^2\phi - 2mn\tau}{\sigma_{\text{s}} + 2\sigma_{\text{b}}}\right]^{1/2} \tag{4.6}$$

provided, of course, that $1 \leq p_{\text{opt}} \leq n$. It should be noted that in order for the expression inside the square root to be positive, we must have $mn^2\phi > 2mn\tau$. We will see in Chapter 7 that $\phi > \tau$ and since n is large, this condition holds. To show that p_{opt} is indeed a unique minimum, we note that

$$\frac{\partial^2 T_{\text{tot}}}{\partial p^2} = \frac{4mn^2\phi - 8mn\tau}{p^3}$$

is also positive exactly when the condition above is met.

CHAPTER 5

Parallel Reorthogonalization

The reorthogonalization step is accomplished by calculating the QR decomposition of the matrix Q_ν, in which the columns of the matrix Q form an orthonormal basis for the column space of Q_ν. The major steps in the calculation are as follows [9]:

- Calculate ${Q_\nu}^{\mathrm{T}} Q_\nu$, which is a symmetric positive definite matrix.
- Compute the Cholesky factorization ${Q_\nu}^{\mathrm{T}} Q_\nu = R^{\mathrm{T}} R$ to determine R, a nonsingular upper-triangular matrix.
- Solve $QR = Q_\nu$ for Q.

Note that mathematically the Q matrix calculated by this algorithm has orthonormal columns:

$$
\begin{aligned}
Q^{\mathrm{T}} Q &= (Q_\nu R^{-1})^{\mathrm{T}} (Q_\nu R^{-1}) = R^{-\mathrm{T}} {Q_\nu}^{\mathrm{T}} Q_\nu R^{-1} = R^{-\mathrm{T}} R^{\mathrm{T}} R R^{-1} \\
&= I_m.
\end{aligned}
$$

Numerically, however, this method has little to recommend it. The columns of Q may be far from orthonormal if $\kappa({Q_\nu}^{\mathrm{T}} Q_\nu)$ is large [9, 10]. On the other hand, the upper bound (2.3) should alert one to the need for reorthogonalization before the columns of Q_ν become too badly behaved. Therefore, Q_ν should not stray too far from orthogonality and, although $\kappa({Q_\nu}^{\mathrm{T}} Q_\nu)$ will not

```
DISTRIBUTE:   for k in p − i − 1 downto 1 loop
                  if node i is the master then
                      read Q_k^(v) from memory;
                      send Q_k^(v) to clockwise neighbor;
                  else
                      receive Q_k^(v) from counter-clockwise neighbor;
                      send Q_k^(v) to clockwise neighbor;
                  end if;
              end loop DISTRIBUTE;
              if node i is the master then
                  read Q_i^(v) from memory;
              else
                  receive Q_i^(v) from counter-clockwise neighbor;
              end if;

              CrossProduct_i ← (Q_i^(v))^T Q_i^(v);
              Sum ← 0;
SUM:          for k in i wrap around p − 1 to i − 1 loop
                  Sum ← Sum + CrossProduct_k;
                  send CrossProduct_k to counter-clockwise neighbor;
                  receive CrossProduct_k from clockwise neighbor;
              end loop SUM;
              compute Cholesky factor R using Sum;
              solve Q_i×R = Q_i^(v) for Q_i;

COLLECT:      if node i is the master then
                  place Q_i in memory;
              else
                  send Q_i to counter-clockwise neighbor;
              end if;
              for k in 1 to p − i − 1 loop
                  receive Q_k from clockwise neighbor;
                  if node i is the master then
                      place Q_k in memory;
                  else
                      send Q_k to counter-clockwise neighbor;
                  end if;
              end loop;
```

Figure 4.

Algorithm for Parallel Reorthogonalization.

be precisely equal to 1, it should be tolerably close. Furthermore, since the

columns of Q are generated as solutions of the equation $QR = Q_v$, a backward error rounding analysis can be used to show that if the computations are done in δ-digit decimal arithmetic, then there exists an error matrix $E \in \mathbb{C}^{n\times n}$ of order $10^{-\delta}\|Q_v\|$ such that $QR = Q_v + E$ [9]. These facts, along with the simplicity of the algorithm, justify its use on a parallel computer architecture.

Figure 4 gives the algorithm used to calculate the QR decomposition on node *i*. The decomposition is calculated on the entire ring. A simple analysis of this algorithm finds the following costs for each calculation:

$$({Q_i}^{(\nu)})^{\mathrm{T}}{Q_i}^{(\nu)}: \quad \frac{nm^2\phi}{2p}$$

$$\textstyle\sum_k {Q_k}^{(\nu)}: \quad \frac{1}{2}pm^2\phi^+$$

$$\text{Calculation of } R: \quad \tfrac{1}{6}m^3\phi$$

$$\text{Calculation of } Q_i: \quad \frac{nm^2\phi}{2p}$$

where ϕ^+ is the time required to perform a floating point summation. Adding these times gives

$$T_{\mathrm{cal}} = \left[\frac{nm^2}{p} + \frac{m^3}{6}\right]\phi + \tfrac{1}{2}pm^2\phi^+ \tag{5.1}$$

The factor of $\frac{1}{2}$ in the first term is due to the fact that $({Q_i}^{(\nu)})^{\mathrm{T}}{Q_i}^{(\nu)}$ is symmetric and only half of it need be computed. The communication requires the same block distribution and collection times (T_1 and T_3) as the AQ algorithm

plus a cost of $(p-1)\left[\sigma_{\mathrm{s}}+\frac{1}{2}m(m+1)\tau\right]$ for the $\sum_k Q_k{}^{(\nu)}$ calculation, giving

$$T_{\mathrm{com}} = 2(p-1)\left(\sigma_{\mathrm{b}}+\frac{mn}{p}\tau\right)+(p-1)\left[\sigma_{\mathrm{s}}+\tfrac{1}{2}m(m+1)\tau\right] \qquad (5.2)$$

We may then find p_{opt} by adding (5.1) and (5.2), differentiating,

$$\frac{\partial T_{\mathrm{tot}}}{\partial p} = \frac{2mn\tau - nm^2\phi}{p^2} + \tfrac{1}{2}m^2\phi^{+} + 2\sigma_{\mathrm{b}} + \sigma_{\mathrm{s}} + \tfrac{1}{2}m(m+1)\tau$$

and obtaining

$$p_{\mathrm{opt}} = \left[\frac{2nm^2\phi - 4mn\tau}{m^2\phi^{+} + 4\sigma_{\mathrm{b}} + 2\sigma_{\mathrm{s}} + m(m+1)\tau}\right]^{1/2}. \qquad (5.3)$$

To guarantee a non-negative radicand, we must have $2nm^2\phi > 4mn\tau$ and we will see in Chapter 7 that this condition is easily met. Once again, the second derivative

$$\frac{\partial^2 T_{\mathrm{tot}}}{\partial p^2} = \frac{2nm^2\phi - 4mn\tau}{p^3}$$

is positive exactly when the condition above is met, so p_{opt} is a unique minimum.

CHAPTER 6

Parallel Reorthogonalization with Binary Summation

The *SUM* loop of Figure 4, which accounts for the $\frac{1}{2}pm^2\phi^+$ term in T_{cal}, can be replaced by a binary summation algorithm that can reduce this to a term which is logarithmic in p. The idea of binary summation (or "pairwise summation") is simple. Odd-numbered processors send their cross products to the next even-numbered processors, each of which adds the incoming cross product to its own. These even-numbered processors then pair off to repeat the process among themselves. The binary summation is continued until one processor remains, the one containing the total sum. An example is shown in Figure 5 for $p = 8$. The large circles (⊙) indicate processors that are sending or accumulating partial sums, while the small circles (•) indicate those that are merely passing data along without processing it.

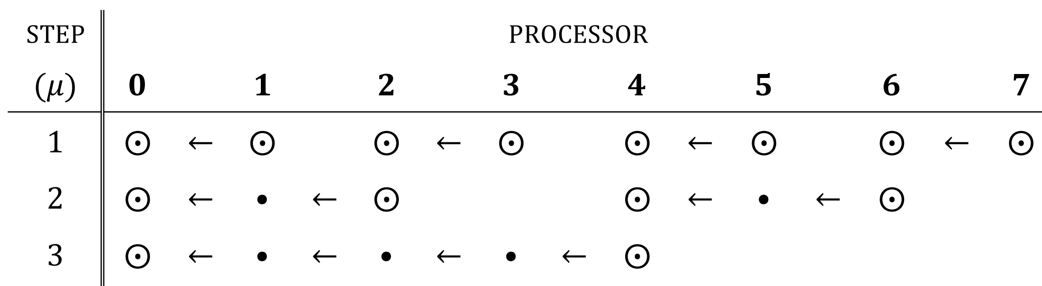

Figure 5.

Binary Summation Example for 8 Processors.

It is evident that the left-hand processors are doing more data transmissions than they would be with the normal linear summation method. Furthermore, the binary algorithm reduces the summation term in the calculations to $\frac{1}{2}m^2(\log_2 p)\phi$, but since only the first processor will have the entire sum, only it can calculate R. It must therefore redistribute R along the ring, allow each processor to calculate its own block of Q, and collect the blocks when they are ready. These facts imply that the algorithm is best suited to machines whose transmission costs are relatively inexpensive compared to calculation costs. To summarize, we give the following calculation costs for each step of the calculation:

$$({Q_i}^{(\nu)})^{\mathrm{T}}{Q_i}^{(\nu)}: \quad \frac{nm^2\phi}{2p}$$

$$\textstyle\sum_k {Q_k}^{(\nu)}: \quad \frac{1}{2}m^2(\log_2 p)\phi^+$$

$$\text{Calculation of } R: \quad \tfrac{1}{6}m^3\phi$$

$$\text{Calculation of } Q: \quad \frac{nm^2\phi}{2p}$$

To figure the transmission time, we need to consider how long it takes for data generated by the last node in the ring to reach processor 0. It is evident from Figure 5 that one transmission time is required at step 1, two at step 2, and four at step 3. In general, it still requires $p-1$ transmission times for data from node $p-1$ to make it all the way around the ring. So the total transmission costs in this algorithm are as follows:

1. Distribute the blocks of Q_v around the ring of processors with cost

$$T_1 = (p-1)\left(\sigma_{\mathrm{b}} + \frac{mn}{p}\tau\right)$$

2. Receive partial sums from the clockwise neighbor with cost

$$T_2 = (p-1)\left[\sigma_{\mathrm{s}} + \tfrac{1}{2}m(m+1)\tau\right]$$

3. Distribute R around the ring of processors with cost

$$T_3 = (p-1)\left[\sigma_{\mathrm{b}} + \tfrac{1}{2}m(m+1)\tau\right]$$

4. Collect the blocks of the new Q matrix from the processors in the ring with cost

$$T_4 = (p-1)\left(\sigma_{\mathrm{b}} + \frac{mn}{p}\tau\right)$$

so that

$$T_{\mathrm{com}} = 2(p-1)\left(\sigma_{\mathrm{b}} + \frac{mn}{p}\tau\right) + (p-1)\left[\sigma_{\mathrm{b}} + \tfrac{1}{2}m(m+1)\tau\right] + (p-1)\left[\sigma_{\mathrm{s}} + \tfrac{1}{2}m(m+1)\tau\right]$$

The deceptive simplicity of this analysis belies some computational problems in the implementation. Notice that each processor *i* in the chain goes through three distinct phases during the execution of the summation:

- If *i* is not odd, add some number Σ_i of partial sums to $CrossProduct_i$
- Send the accumulated partial sum counter-clockwise
- Pass some number Π_i of partial sums counter-clockwise in a bucket brigade fashion.

The task at hand is to determine

- Σ_i, the number of additions processor *i* must calculate and
- Π_i, the number of times processor *i* must pass results after it has sent its own accumulated partial sum counter-clockwise.

Furthermore, we must address the case of a binary summation in which the number of processors, *p*, is not an integral multiple of two. We will begin with the calculations of Σ_i and Π_i for which $p = 2^k$ for some integer *k*.

The calculation of Σ_i is straightforward and is shown in Figure 6.

```
        Σ_i ← 0;
        π_i ← if node i is the master then p else i;

COUNT:  while π_i is even loop
              Σ_i ← Σ_i + 1;
              π_i ← π_i/2;
        end loop COUNT;
```

Figure 6.

Algorithm for Calculation of Σ_i when $p = 2^k$.

The value π_i reflects the effective processor number at each summation and is initialized to the processor number, *i*, except when *i* is the master processor, in which case it is set to *p* to compensate for the indexing. At each iteration, π_i is halved and *i* is the left-hand member of a processor pair as long as π_i is even. Then Σ_i merely counts the number of times that processor *i* will

perform an addition. Note that the master processor will have $\Sigma_0 = \log_2 p = k$, the desired value.

The problem of calculating Π_i is more complicated. Note that in Figure 5, the intervals (0,3) and (4,7) are duplicates for $\mu = 1, 2$. But when $\mu = 3$ and processor 0 receives its final partial sum from processor 4, processors 1, 2, and 3 must pass it along since they reside in the left-hand interval (0, 3). Similarly, the subintervals (0,1) and (2,3) are identical for $\mu = 1$. But processor 1 must pass along a partial sum from processor 2 because it is in the left-hand subinterval (0,1). If we generalize and continue to subdivide the intervals in this manner, we see that the number of passes processor *i* must make

```
            Πi ← 0;
            if node i is not the master then
                Left ← 0;
                Right ← p − 1;
DIVIDE:         while Left ≠ Right loop
                    Midpoint ← ⌊Right/2⌋;
                    if Left ≤ i < Midpoint then
                        Πi ← Πi + 1;
                        Right ← Midpoint;
                    else
                        Left ← Midpoint + 1;
                    end if;
                end loop DIVIDE;
            end if;
```

Figure 7.

Algorithm for Calculation of Π_i when $p = 2^k$.

is equal to the number of times it is in a left-hand subinterval. The algorithm is given in detail in Figure 7.

We now turn our attention to the case when $p \neq 2^k$. We will divide the chain into partitions, each of which contains a number of processors which is an integral power of two. Starting on the left-hand side of the chain, we determine the largest power that is less than or equal to p. This number is the length of the left-most partition. We then repeat the process to obtain the remaining partitions. An example is shown for $p = 7$ in Figure 8. The proces-

STEP	PARTITION 1							PARTITION 2				PARTITION 3	
(μ)	**0**		**1**		**2**		**3**		**4**		**5**		**6**
1	⊙	←	⊙		⊙	←	⊙		⊙	←	⊙		⊙
2	⊙	←	•	←	⊙				⊙	←	•	←	⊙
3	⊙	←	•	←	•	←	•	←	⊙				

Figure 8.

Binary Summation Example for 7 Processors.

sors in each partition essentially execute a binary summation using the values obtained by the algorithms in Figures 6 and 7. The only modifications required are finding and using the left and right endpoints of the partition containing node *i*.

A glance at Figure 8 indicates that a knowledge of the left and right

endpoints of the partition is required not only to calculate the length of the partition but also to determine whether or not node i should expect partial sums from a neighboring partition to the right. The algorithm consists merely of searching for the partition whose length is less than p and checking if node i lies within. If not, we find the left-most partition in the remaining processors and perform the check again. We continue until the partition is found. The algorithm is given in Figure 9.

```
End ← 0;
Remaining ← p;
Interval ← maximum number of processors allowed;
loop
        while ⌊Remaining/Interval⌋ = 0 loop
                Interval ← ⌊Interval/2⌋;
        end loop;
        exit when End ≤ i < End + Interval;
        Remaining ← Remaining − Interval;
        End ← End + Interval;
end loop;
Left ← End;
Right ← End + Interval − 1;
```

Figure 9.

Algorithm for Calculation of the Endpoints of a Partition.

The new calculation of Σ_i is simple. We begin by calculating the left

```
calculate Left and Right endpoints of partition containing node i using Figure 9;
if i = Left then
        Σ_i ← if i is in the right-most partition then 0 else 1;
        π_i ← Right − Left + 1;
else
        Σ_i ← 0;
        π_i ← i − Left + 1;
end if;
update Σ_i and π_i using the COUNT loop of Figure 6;
```

Figure 10.

Algorithm for the General Calculation of Σ_i.

and right endpoints of the partition containing node *i* so that π_i now contains the number of processors between node *i* and the left end of the partition.

```
calculate Left and Right endpoints of partition containing node i using Figure 9;
Π_i ← 0;
Extra ← (Right = p − 1);
if node i is not the master then
        update Π_i, Right, and Left using the DIVIDE loop of Figure 7;
end if;
if Extra then Π_i ← Π_i + 1 end if;
```

Figure 11.

Algorithm for the General Calculation of Π_i.

We then continue exactly as before, counting the number of times node *i* will

perform an addition. The only remaining change is the case for which node *i* is the left-most node in a partition which is not the right-most partition. In this case, node *i* will receive a partial sum from a neighboring right partition and must perform one extra summation (see Figure 8). The algorithm for this process is given in Figure 10. The calculation of Π_i also follows easily from Figures 7 and 9 and is shown in Figure 11.

CHAPTER 7

The Empirical Calculation of ϕ, σ, and τ

The data for this thesis was collected on a parallel machine called McMob, a system built and programmed by the University of Maryland Institute for Advanced Computer Studies. It consists of sixteen Motorola MC68000 microprocessors, each with one megabyte of memory and connected to its neighbors by an electronic conveyor belt. Transmission of data is performed sixteen bits at a time. Outgoing integers are loaded into an outbound buffer register which becomes clear when the data has been successfully sent. Incoming data interrupts the processor, which queues the data until an entire message has been received.

```
NUMITS ← some large number;
start  ← clock();
for k in 1 to NUMITS loop
        perform floating point calculation;
end loop;
φ  ← (clock() − start)/NUMITS;
```

Figure 12.

Algorithm for the Empirical Calculation of ϕ.

All floating point operations on McMob are performed by software emulation. The calculation of ϕ is straightforward and is given in Figure 12. Here, we use a function, *clock*, that returns the amount of time used by the program up to the invocation of the function. We have also left the definition of the floating point calculation purposefully vague since the exact calculation will depend on the particular computational example being performed (see Chapter 8). Since the definition of ϕ^+ is simply the time required to perform a floating point summation, however, it is independent of the computational example and was calculated to be 112 microseconds. The time required to perform a floating point multiplication and addition was found to be 415 microseconds.

```
total ← 0;
NUMITS ← some large number;
for k in 1 to NUMITS loop
        start ← clock();
        if this is processor 1 then
                send block to processor 2;
                wait for block from processor 2;
        else
                wait for block from processor 1;
                send block to processor 1;
        end if;
        total ← total + clock() − start;
end loop;
total ← total/(2 × NUMITS);
```

Figure 13.

Empirical Calculation Loop for σ_b.

On the other hand, the calculations of σ and τ require some knowledge of the transfer being done. During the block distribution phases of the parallel procedures, each block of the matrix is passed along to the correct processor in a bucket brigade fashion. That is, processor i sends a block to processor $i+1$, which is waiting to receive it. At the next step, processor $i+1$ sends the block to processor $i+2$ while processor i receives from processor $i-1$. This sort of transfer leads to the algorithm for determining transmission times of blocks between two processors shown in Figure 13. The correction factor of $2 \times NUMITS$ takes into account that two block transmissions are occurring at each iteration of the loop and the loop is executed $NUMITS$ times.

On the other hand, a different scenario arises when the blocks of the matrix are being passed around the ring as part of the block multiplication

```
total ← 0;
NUMITS ← some large number;
for k in 1 to NUMITS loop
        start ← clock();
        send block to neighbor;
        receive block from neighbor;
        total ← total + clock() − start;
end loop;
total ← total/(2 × NUMITS);
```

Figure 14.

Empirical Calculation Loop for σ_s.

given in Figure 3. In that case, processor i is sending a block to processor $i+1$ at precisely the same time as processor $i-1$ is sending a block to processor i. The interrupts generated by the incoming data steal time from processor i, which is trying to get data out. This situation should inflate the value of σ_s but should have no effect on τ. The algorithm for this calculation is given in Figure 14.

In order to get the most accurate results possible, each of these algo-

Size (ints)	Time (μsecs)	
	Bucket Brigade	Simultaneous Exchange
1	6669	13338
101	49681	56586
201	93197	99836
301	136447	143352
401	179963	186334
501	223213	229852
601	266463	273374
701	309717	316674
801	353232	359898
901	396480	403182

Figure 15.

Time Used in the Determination of σ_s, σ_b, and τ.

rithms was performed for block sizes ranging from 1 to 901 integers with the results shown in Figure 15. The block sizes and their associated execution times was then fit to a line using linear least squares. The slope of the resulting line is therefore τ and the y-intercept is σ. The value of σ_b was deter-

mined to be 6135 microseconds and the value of σ_s was calculated to be 12851 microseconds. The value of τ in each case was 433 microseconds. C source code for the determinations of σ and τ can be found in the file *time.c* in Appendix 2.

CHAPTER 8

Results of the Parallel AQ Algorithm

We will take as an example of the parallel multiplication algorithm a matrix A defined by

$$A = Z + uu^{\mathrm{T}}$$

where $Z \in \mathbb{C}^{n \times n}$ is a random diagonal matrix (stored as a one-dimensional array) and $u \in \mathbb{C}^{n}$ is a random vector. The calculation required to determine an element α_{ij} of A is therefore

$$\alpha_{ij} = \upsilon_i \upsilon_j + \delta_{ij}\zeta_i$$

where δ_{ij} is the Kronecker delta function defined by

$$\delta_{ij} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

and the calculation required to determine an element q'_{ij} of $Q' = AQ$ is the usual inner product

$$q'_{ij} = \sum_{k=1}^{n} \alpha_{ik} q_{kj} \,.$$

Thus the floating point calculation central to this computational example is

$$q_{ij} + (\upsilon_i \upsilon_k + \delta_{ik}\zeta_i) q_{kj}.$$

The source code for the random diagonal matrix calculation is given in the

file *diag.c* in Appendix 2.

Figure 16 shows the values obtained for the several processor config-

p	Obtained (sec)				Expected (sec)			
	T_1	T_2	T_3	T_{cal}	T_1	T_2	T_3	T_{cal}
2	0.09	0.38	0.09	17.99	0.09	0.19	0.09	17.64
3	0.13	0.33	0.13	12.34	0.13	0.21	0.13	11.76
4	0.14	0.56	0.24	8.96	0.15	0.22	0.15	8.82
5	0.15	0.63	0.32	7.20	0.16	0.23	0.16	7.06
6	0.17	0.69	0.35	6.17	0.18	0.24	0.18	5.88

Figure 16.

Results of Random Diagonal Matrix Example

with $n = 100$, $m = 4$, and $\phi = 882\mu$secs.

urations and the expected values from (4.1) and (4.5). There is a marked discrepancy between the expected and obtained values for T_2, the block rotation time during the summation loop, and T_3, the block collection time of the product. An independent experiment that sent blocks of data around the ring of processors on McMob revealed that timing data *transmissions* from a DOMINO node was stable and predictable. Timing data *receptions* on a node, however, was very erratic, and T_2 and T_3 involve this type of reception. Although a number of solutions to the problem were attempted, none yielded any satisfactory explanation for the behavior.

Nonetheless, the total expected values and obtained results are given in Figure 17.

p	**Obtained T_{tot} (sec)**	**Expected T_{tot} (sec)**
2	18.55	18.01
3	12.93	12.23
4	9.90	9.34
5	8.30	7.61
6	7.38	6.48

Figure 17.

Total Costs for Random Diagonal Matrix Example.

Clearly, the effect of adding more processors to the computations is beneficial. This can be explained by the overwhelming dominance of the calculation time, T_{cal}. But this is what should be expected from an examination of (4.4) and (4.5); T_{cal} is $O(n^2)$ whereas T_{com} is only $O(n)$. Furthermore, the large value of ϕ for this matrix assures that the calculation time will be the costliest part of the algorithm.

The example demonstrates the effectiveness of the parallel algorithm on large full matrices where many floating point calculations are necessary to determine the q'_{ij}. There are, however, applications for which the parallel method is not useful. We will take as a counter-example of the computational properties of the parallel algorithm a random walk on a $\gamma \times \gamma$ triangular grid.

This example will illustrate the ineffectiveness of the parallel algorithm on large sparse matrices.

Figure 18 illustrates a grid for $\gamma = 5$.

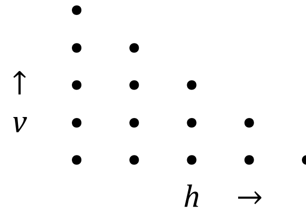

Figure 18.

A 5 × 5 Triangular Grid.

The points of the grid are labeled

$$(v, h) \qquad h = 1, 2, \cdots, \gamma; \quad v = 1, 2, \cdots, \gamma - h + 1$$

From any arbitrary point $(\hat{v}, \hat{h})$ on the grid a jump may occur to the north at $(\hat{v} + 1, \hat{h})$, to the east at $(\hat{v}, \hat{h} + 1)$, to the south at $(\hat{v} - 1, \hat{h})$, or to the west at $(\hat{v}, \hat{h} - 1)$. The probability of jumping to the south or to the west is

$$P_{\mathrm{SW}}(v, h) = \begin{cases} \frac{1}{2}\left[\dfrac{v + h - 2}{\gamma - 1}\right] & \text{if both points exist} \\ \dfrac{v + h - 2}{\gamma - 1} & \text{otherwise} \end{cases} \tag{9.1}$$

so the probability of jumping to the north or the east is

$$P_{\mathrm{NE}}(v, h) = \begin{cases} \frac{1}{2}\left[1 - \dfrac{v + h - 2}{\gamma - 1}\right] & \text{if both points exist} \\ 1 - \dfrac{v + h - 2}{\gamma - 1} & \text{otherwise} \end{cases} \tag{9.2}$$

For example, $P_{\mathrm{NE}}(1,2) = \frac{3}{8}$ since both the north point, (2,2), and the east point, (1,3), are on the grid. However, $P_{\mathrm{SW}}(1,2) = \frac{1}{4}$ since the west point, (1,1), is on the grid, but the south point does not exist.

If we number the points on the grid consecutively from 1 to $\frac{1}{2}(\gamma^2 + \gamma)$ then the random walk can be expressed as a finite Markov chain whose stochastic matrix A consists of the probabilities α_{ij} of jumping from point j to point i. We may then calculate the steady state probabilities of the chain, contained in the eigenvector corresponding to the eigenvalue unity [5]. Stewart [12] points out that the chain is cyclic with period two and hence A has an eigenvalue of -1 as well as $+1$.

q_{5j}				
q_{4j}	q_{9j}			
q_{3j}	q_{8j}	$q_{12,j}$		
q_{2j}	q_{7j}	$q_{11,j}$	$q_{14,j}$	
q_{1j}	q_{6j}	$q_{10,j}$	$q_{13,j}$	$q_{15,j}$

Figure 19.

Mapping of the Points of the Grid to q_j for $\gamma = 5$.

The points of the grid are matched with the q_j as shown in Figure 19. The matrix A is never explicitly used; to calculate $(Aq_j)_i$, the ith element of Aq_j, one need only regard the components of q_j as the average number of individuals at the points of the grid and use (9.1) and (9.2) to calculate the

number of individuals that will be at point *i* at the next transition [12]. If we let $q_{ij}^{[d]}$, $d \in \{\mathrm{N, S, E, W}\}$, be the number of individuals in the north, south, east, and west, respectively, and assuming that point *i* has coordinates $(\hat{v}, \hat{h})$, then

$$(Aq_j)_i = q_{ij}^{[\mathrm{N}]} P_{\mathrm{NE}}(\hat{v}, \hat{h}) + q_{ij}^{[\mathrm{S}]} P_{\mathrm{SW}}(\hat{v}, \hat{h}) + q_{ij}^{[\mathrm{E}]} P_{\mathrm{NE}}(\hat{v}, \hat{h}) + q_{ij}^{[\mathrm{W}]} P_{\mathrm{SW}}(\hat{v}, \hat{h}) \quad (9.3)$$

Hence the calculation of $(Aq_j)_i$ requires a knowledge of the number of individuals at each neighbor of q_{ij}. This presents a complication in the parallel computation since the q_{dj} will be circulating around the ring of processors and each AQ node must keep track of when to multiply an element of a block by a probability and add it to the $(Aq_j)_i$. Therefore, every go node calculates the index of each neighbor for every point *i* on the grid that falls within the node's block boundary as part of the initialization procedure and supplies the AQ node with this data through auxiliary storage. It is then a simple matter for the AQ node to check each incoming block for any $q_{ij}^{[d]}$ it requires. The source code for the implementation of this example is given in the file *markov.c* in Appendix 2.

Execution on the McMob processors, however, gave the results shown in in Figure 20.

p	T_{com} (sec)	T_{cal} (sec)	T_{tot} (sec)
2	0.26	0.50	0.76
3	0.58	0.36	0.94
4	0.72	0.32	1.04
5	0.82	0.29	1.11

Figure 20.

Results of Markov Chain Matrix Calculation
with $\gamma = 10, n = 55$, and $m = 4$.

The disappointing performance can be explained by the fact that each element of the Aq_j is determined by performing only the four floating point operations given in (9.3), so the number of calculations to perform in order to compute an entire block of Q is $\frac{4mn}{p}$. But the time required to send a block to its neighbor during the calculation step increases linearly in n (see (4.5)). Since the calculation time T_{cal} is also linear in n it does not dominate the block multiplication step as it did in the random diagonal matrix example. Therefore, adding more processors to the problem only increases its cost.

CHAPTER 9

Results of the Parallel Reorthogonalization Algorithm

The parallel reorthogonalization algorithm was run as part of the two computational examples described in the previous chapter. Results were obtained for both the linear and binary summations and are discussed here.

We shall use for comparison the results obtained for the diagonal matrix of degree $n = 100$ and subspace dimension $m = 4$. The execution times for the calculation of $Q^{\mathrm{T}}Q$, the Cholesky factorization of $Q^{\mathrm{T}}Q$ for R , and the solution for the new value of Q do not depend on the type of summation used and are shown in Figure 21.

p	**Obtained (sec)**			**Expected (sec)**		
	$Q^{\mathrm{T}}Q$	Q	R	$Q^{\mathrm{T}}Q$	Q	R
2	0.21	0.051	0.27	0.17	0.004	0.17
3	0.14	0.051	0.18	0.11	0.004	0.11
4	0.10	0.051	0.13	0.08	0.004	0.08
5	0.08	0.051	0.11	0.07	0.004	0.07
6	0.07	0.051	0.09	0.06	0.004	0.06

Figure 21.

Results of Summation-Independent Calculations.

The discrepancy between the obtained and expected values for the Cholesky factorization can be explained by the fact that the expected value takes into account only the inner product calculation central to the factorization. Indeed, for a large matrix this inner product would consume most of the time. We, on the other hand, are dealing with a 4×4 matrix and the other floating point operations in the algorithm take their toll. In any case, the Cholesky factorization does not contribute significantly to the total computational cost.

Figure 22 shows the transmission costs.

p	**Obtained (sec)**			**Expected (sec)**		
	T_1	T_2	T_3	T_1	T_2	T_3
2	0.09	0.01	0.09	0.09	0.01	0.09
3	0.12	0.10	0.13	0.13	0.03	0.13
4	0.14	0.24	0.20	0.15	0.04	0.15
5	0.15	0.35	0.24	0.16	0.05	0.16
6	0.17	0.47	0.28	0.18	0.07	0.18

Figure 22.

Transmission Costs for the Parallel Reorthogonalization.

Once again the erratic performance of the T_2 and T_3 values shows itself but we are still communicating small blocks around the ring. Unfortunately, the executable code for the implementation is very large and memory constraints on the McMob processors precluded testing with larger blocks. Although the dominance of the floating point operations in the AQ algorithm

masked the problem, we are not as fortunate here. Data transmission costs and calculation costs are on the same order of magnitude and the fluctuations have a decided impact. The totals are given in Figure 23 and show roughly the same optimal number of processors, so it is safe to say that the parallel algorithm is worthwhile, although not nearly to the degree that the AQ algorithm is.

p	**Obtained T_{tot} (sec)**	**Expected T_{tot} (sec)**
2	0.74	0.53
3	0.72	0.51
4	0.86	0.50
5	0.98	0.51
6	1.13	0.55

Figure 23.

Total Costs for the Reorthogonalization Algorithm.

The execution times of the linear and binary summations during the accumulation of the $({Q_i}^{(\nu)})^{\mathrm{T}}{Q_i}^{(\nu)}$ are shown side-by-side in Figure 24. The low values for the calculation of the QR summation in both cases are due to the fact that a floating point summation is being performed as the critical floating point calculation of the procedure. Recall that a floating point summation can be done on the McMob computer in only 112 microseconds, compared to the 415 required for a floating point multiplication and addition or

p	Linear (sec)		Binary (sec)	
	Calculation	Transmission	Calculation	Transmission
2	0.0011	0.19	0.0011	0.31
3	0.0023	0.41	0.0120	0.51
4	0.0034	0.62	0.0023	0.85
5	0.0046	0.85	0.0132	1.07
6	0.0057	1.06	0.0034	1.23

Figure 24.

Results of the Linear and Binary Summations.

the roughly 13,000 required to send one integer from a processor to its neighbor. This fact ensures that the communication time will far exceed the calculation time when the invariant subspace being calculated is small. Note that the time required to calculate the summation in the binary algorithm has been reduced for most processor configurations. This reduction is due to the fact that processor 0 (the one accumulating the final sum) is performing fewer additions than it would be if the summation were linear. But considering the magnitude of the contribution of the summation to the total cost, it does not seem advantageous to use the binary summation algorithm. This conclusion is especially true since we have not even considered the additional time needed for the recirculation of R around the ring of processors in the above results.

CHAPTER 10

Conclusions

This thesis successfully implements a simultaneous iteration algorithm for eigenvalue and eigenvector calculation in a parallel environment. The algorithm is based on the theory that nested invariant subspaces of a large matrix can be used to determine the largest absolute eigenvalues of that matrix. Its main benefit is that the algorithm requires the user to supply only a procedure for calculating a matrix multiplication. This fact leads to a modular and flexible software design that also lends itself to a parallel solution.

We have seen that the parallel calculation of this matrix multiplication is very worthwhile if the process used to calculate the elements of the product is computationally expensive. Since traditional matrix multiplication algorithms typically involve the accumulation of inner products, this requirement is easily satisfied for many applications. The parallel method is almost certainly useless for large sparse matrix problems if access to the individual elements is inexpensive relative to the cost of data transmission.

Since our method requires the periodic reorthogonalization of the column vectors that span the invariant subspace due to the fact that our ma-

trix is not in general Hermitian, the thesis implements this process in parallel as well. In contrast to the costly arithmetic required to calculate the matrix product, the reorthogonalization requires only inexpensive floating point summations in the parallel sections of the code. This situation gives far greater emphasis to the transmission costs but the results demonstrate that it can still be profitable to perform the reorthogonalization in parallel. The data do not, however, recommend the use of the binary summation method. Although the summation time is slightly less with the binary algorithm, the additional time required for transmission negates this benefit.

APPENDIX 1

Table of Symbols

Symbol	Description	Page
A	An $n \times n$ matrix whose nested invariant subspaces are calculated by this thesis	1
A_{ik}	The (i,k) block of the matrix A	13
α_{ij}	The (i,j) element of the matrix A	6
α'_{ij}	The (i,j) element of the matrix used in the random diagonal matrix computational example of Chapter 8	36
B	An $m \times m$ matrix used in the SRR step	6
$\mathbb{C}^{n \times m}$	The set of $n \times m$ complex matrices	1
$\mathbb{C}^n$	The set of n-dimensional complex-valued vectors	1
δ	The number of decimal digits that can be allowed to be lost between reorthogonalizations	5
δ_{ij}	The Kronecker delta function	36
E	An $n \times n$ error matrix	7

Symbol	Description	Page
ϕ	The time necessary to execute a floating point multiplication and addition, along with indexing and looping overhead	15
ϕ^+	The time necessary to execute a floating point summation, along with any indexing and looping overhead	20
$\mathbf{\Gamma}$	The invariant subspace of A	38
$\mathbf{\Gamma}_r$	The rth dominant invariant subspace of A	3
γ	The dimension of the triangular grid used in the computational example of Chapter 8	38
η	An upper bound on the number of iterations that can be performed safely with $R_{\nu+1} = I_m$	5
I_m	The $m \times m$ identity matrix	5
$\kappa(T)$	The condition number of T with respect to inversion	5
Λ_A	The set of eigenvalues of A	1
λ_i	The ith eigenvalue of A	1
m	The column dimension of Q	1
μ	A step in the binary summation algorithm	22
n	The degree of A	1

Symbol	Description	Page
ν	A step in the basic iteration	4
P_d	A probability function used in the computational example of Chapter 8 ($d \in \{\mathrm{NE}, \mathrm{SW}\}$)	39
p	The number of processors used	12
p_{opt}	The optimal number of processors to use with a parallel algorithm	17
π_i	A parameter used in the calculation of Σ_i	25
Π_i	The number of passes a node must perform in the binary summation algorithm	25
Q	A matrix whose columns form an orthonormal basis for the nested invariant subspaces of A	1
$Q^{\lvert r}$	The first r columns of Q	3
${Q_i}^{(\nu)}$	The ith block of Q_ν	13
Q_ν	The matrix Q at the νth iteration	4
q_j	The jth column of Q	1
$q_{ij}^{[d]}$	The number of individuals at neighbor d in the computational example of Chapter 7 ($d \in \{\mathrm{N}, \mathrm{S}, \mathrm{E}, \mathrm{W}\}$)	41

Symbol	Description	Page
R_ν	The matrix R at the νth simultaneous iteration	5
r_i	The ith column of the residual matrix	7
ρ_ν	A nonzero scaling factor used in the power method	4
$\mathbf{\Psi}$	An arbitrary m-dimensional subspace	4
Σ_i	The number of summations node i must perform in the binary summation algorithm	25
σ_{b}	The startup time required in communication between processors when the processors are sending data in a bucket brigade fashion	16
σ_{s}	The startup time required in communication between processors when the processors are simultaneously exchanging data	16
T	An $m \times m$ matrix whose eigenvalues are the same as the m-dimensional invariant subspace of A	1
$T^{\lceil r}$	The r-dimensional leading principal submatrix of T	3
T_{cal}	The calculation time required by an algorithm	15
T_{com}	The communication time required by an algorithm	15
T_{tot}	The total cost of an algorithm	15

Symbol	Description	Page
τ	The transmission rate of data	15
u	An n-dimensional random vector	36
υ_i	An element of $u \in \mathbb{C}^n$	36
Z	An $n \times n$ random diagonal matrix stored as a one-dimensional array	36
ζ_i	The ith element of $Z \in \mathbb{C}^{n \times n}$ stored as a one-dimensional array	36

APPENDIX 2

C Source Code for the Implementation

This appendix contains the C source code for the program developed in this thesis. The following table gives a short description of each file.

File	Description
Makefile	Contains a specification for the UNIX *make* program, which builds object and executable files from source code.
aq.h	Contains constants and definitions for the structures used by the nodes that perform the *AQ* algorithm.
debug.h	Contains definitions for debugging facilities used by the routines in the program.
diag.h	Contains the definitions for the random diagonal matrix example used in Chapter 8.
go.h	Contains the definitions necessary for manipulation of the go node auxiliary storage.
markov.h	Contains the definitions for the Markov chain example used in Chapter 8.
matrix.h	Contains definitions for the matrix and vector data types used by the routines in the program.

File	Description
qr.h	Contains the definitions for the auxiliary storage structure used by the nodes that calculate the QR reorthogonalization.
setup.h	Contains constants used for the configuration of the computational network.
time.h	Contains definitions for the clock timer functions necessary for guaging the efficiency of the parallel network.
aq.c	Contains code for the node program which calculates the product of matrices A and Q in a circular fashion.
boot.c	Contains code for the boot node.
diag.c	Contains code for the random diagonal matrix example used in Chapter 8.
go.c	Contains code for the go node.
markov.c	Contains code for the Markov chain example used in Chapter 8.
matrix.c	Contains code for the basic matrix operations, like addition, multiplication, inner product, and Cholesky factorization.
qr.c	Contains code for the node program which calculates the QR reorthogonalization.
send.c	Contains code to time clockwise and counter-clockwise transmission times.

File	Description
srrit.c	Contains the driver and support routines for the SRR iterations. Each of the major routines in this file are C translations of FORTRAN counterparts found in Stewart [12] and Smith [8].
time.c	Contains code for the calculations of σ and τ.

```
#  Make File
#
#
#  AUTHOR:      John R. Meyer
#
#  DATE:        6/18/91 23:21:36
#
#  VERSION:     @(#)Makefile 4.1
#
#
#  DESCRIPTION:
#
#       This file contains the file dependencies used by the UNIX "make"
#       program for the SRR iteration program.
#
#       The contents of this file form part of an appendix to the thesis,
#       "A Parallel Implementation of a Simultaneous Iteration Algorithm
#       for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
#       Matrices", by John R. Meyer and submitted to the faculty of the
#       Graduate School of the University of Maryland in partial fulfillment
#       of the requirements for the degree of master of science.



#  "make" program concerns.

.SUFFIXES :     .o .c .h .c~ .h~ .ln
.KEEP_STATE :
SHELL           = /bin/sh



#  The following program setup definitions are optional.  If not defined
#  here, they will default to values within the header files.
#
#  In setup.h:
#
#       -DNUMVIRPROCS=n         The number of "virtual" processors,
#                               that is, the number simulated on
#                               single-processor machines and the
#                               actual number on parallel machines.
#       -DMAXPROCS=n            The maximum number of processors this
#                               program can handle, which must be a
#                               power of 2.
#       -DVERBOSE=[TRUE|FALSE]  If true, print out program statistics.
#       -DEPSILON=n             The error tolerance to be used in the
#                               tests for convergence.
#       -DMAXIT=n               The maximum number of iterations to
#                               perform.
#
#  In matrix.h:
#
#       -DCOLUMNSOFQ=n          The number of columns of the Q matrix,
#                               the dimension of the invariant subspace
#                               used.
#       -DNUMVECTORS=n          The number of eigenvectors of Q to
#                               calculate.
#       -DITVECTORS=n           The number of vectors of Q with which
#                               to iterate.
#
#  In qr.h:
#
#       -DBINSUM=[TRUE|FALSE]   Determines if a binary summation is to
#                               be performed during the QR reorthogo-
#                               nalization.

SETUP           = -DNUMVIRPROCS=2 -DBINSUM=FALSE


#*********************  McMob definitions  *************************


#  Random diagonal matrix example

DIAG            = diag
DIAG.ln         = $(DIAG).ln
DIAG.c          = diag.c
DIAG.o          = $(DIAG.c:.c=.o)


#  Markov chain example

MARKOV          = markov
MARKOV.ln       = $(MARKOV).ln
MARKOV.c        = markov.c
MARKOV.o        = $(MARKOV.c:.c=.o)

```



```

#  Timer programs

TIMER           = time
TIMER.c         = time.c
TIMER.o         = $(TIMER.c:.c=.o)
SEND            = send
SEND.c          = send.c
SEND.o          = $(SEND.c:.c=.o)


#  DOMINO libraries

DOMDIR          = ./dominomc
DOM.a           = $(DOMDIR)/lmclib.a
PRODIR          = ./dominomc/profacemc
PRO.a           = $(PRODIR)/profacelib.a


#  Archive file

ARFILE          = srr.tar


#  Program listing file

PRFILE          = srr.pr


#  Source and object files common to all computational examples

COM.c           = aq.c boot.c go.c matrix.c qr.c srrit.c
COM.o           = $(COM.c:.c=.o)


#  C compiler

CC              = mcc
CLIBS           = /parallel/mob/mcmob/lib/libc.a /lib/libc.a
MACHDEPS        = -DVAX=FALSE -DMCMOB=TRUE -DSINGLE_PROCESSOR=FALSE
CFLAGS          = -I$(DOMDIR) $(MACHDEPS)


#  Loader

LD              = mcc
LDFLAGS         =


#  Lint program checker

LINT            = lint
LFLAGS          = -bu $(CFLAGS)


#  Object file size reporter

SIZE            = msize


#  Archiver

AR              = tar
ARFLAGS         = cf $(ARFILE)


#  Record compilation errors.

TEE             = tee
ERRFILE         = cc.errors

#*******************  End of McMob definitions  *********************



#***********************  VAX definitions  **************************
#
#
#  Random diagonal matrix example
#
#DIAG           = diag
#DIAG.ln        = $(DIAG).ln
#DIAG.c         = diag.c
#DIAG.o         = diag.o
#
#
#  Markov chain example
```

```
175 #
176 #MARKOV         = markov
177 #MARKOV.ln      = $(MARKOV).ln
178 #MARKOV.c       = markov.c
179 #MARKOV.o       = markov.o
180 #
181 #
182 #  Timer programs
183 #
184 #TIMER          = time
185 #TIMER.c        = time.c
186 #TIMER.o        = time.o
187 #SEND           = send
188 #SEND.c         = send.c
189 #SEND.o         = send.o
190 #
191 #
192 #  DOMINO libraries
193 #
194 #DOMDIR         = ./domino
195 #DOM.a          = $(DOMDIR)/domv.a
196 #PRODIR         =
197 #PRO.a          =
198 #
199 #
200 #  Archive file
201 #
202 #ARFILE         = srr.tar
203 #
204 #
205 #  Program listing file
206 #
207 #PRFILE         = srr.pr
208 #
209 #
210 #  Source and object files common to all computational examples
211 #
212 #COM.c          = aq.c boot.c go.c matrix.c qr.c srrit.c
213 #COM.o          = aq.o boot.o go.o matrix.o qr.o srrit.o
214 #
215 #
216 #  C compiler
217 #
218 #CC             = cc
219 #CLIBS          =
220 #MACHDEPS       = -DVAX=TRUE -DMCMOB=FALSE -DSINGLE_PROCESSOR=TRUE
221 #CFLAGS         = -I$(DOMDIR) $(MACHDEPS)
222 #
223 #
224 #  Loader
225 #
226 #LD             = cc
227 #LDFLAGS        = -lm
228 #
229 #
230 #  Lint program checker
231 #
232 #LINT           = lint
233 #LFLAGS         = -bu $(CFLAGS)
234 #
235 #
236 #  Object file size reporter
237 #
238 #SIZE           = size
239 #
240 #
241 #  Archiver
242 #
243 #AR             = tar
244 #ARFLAGS        = cf $(ARFILE)
245 #
246 #
247 #  Record compilation errors.
248 #
249 #TEE            = tee
250 #ERRFILE        = cc.errors
251 #
252 #********************** End of VAX definitions **********************
253
254
255
256 #  This make file contains specifications for two different computational
257 #  examples:  diag and markov.  To add more examples, perform the
258 #  following procedure:
259 #
260 #       -       Make a header file for the example.  This header file
261 #               must declare the preprocessor macro DEGREEOFA.
```



```
262 #
263 #       -       Make a source file for the example.  The source file
264 #               must contain procedure definitions for the following
265 #               routines:
266 #
267 #                       void aqinit (struct aqaux *);
268 #                       void aqmultiply (struct aqaux *, int, BLOCK *,
269 #                               BLOCK *, int, int);
270 #
271 #               The first routine performs any initialization required
272 #               on the AQ auxiliary structure, while the other actually
273 #               calculates the product of A times Q on a block-by-block
274 #               basis.
275 #
276 #       -       Place the target definitions for the new example in this
277 #               make file under each computer architecture:
278 #
279 #                       NEW_EXAMPLE = new_example
280 #                       NEW_EXAMPLE_SRC = ...
281 #                       NEW_EXAMPLE_OBJ = $(NEW_EXAMPLE_SRC:.c=.o)
282 #
283 #               Include a section describing how to build the new example
284 #               target after the computer architecture definitions.
285 #
286 #       -       Place a group of lines in setup.h that optionally include
287 #               the example's header file:
288 #
289 #                       #ifdef NEW_EXAMPLE
290 #                       #include "new_example_header_file.h"
291 #                       #endif
292 #
293 #       -       Add a conditional macro to the list below:
294 #
295 #                       $(NEW_EXAMPLE) := EXAMPLE = -DNEW_EXAMPLE
296 #
297 #       -       Add the new target to the list of targets to remove
298 #               when making clean.
299
300
301
302 #  Conditional macros based upon what we're building.  On systems that
303 #  do not support conditional macros, uncomment the lines at the end
304 #  of this section and configure them appropriately for the computational
305 #  example desired.
306
307 $(DIAG)         := EXAMPLE = -DDIAG
308 $(DIAG)         := EX.c = $(DIAG.c)
309 $(DIAG)         := EX.o = $(DIAG.o)
310 $(DIAG.ln)      := EXAMPLE = -DDIAG
311 $(DIAG.ln)      := EX.c = $(DIAG.c)
312
313 $(MARKOV)       := EXAMPLE = -DMARKOV
314 $(MARKOV)       := EX.c = $(MARKOV.c)
315 $(MARKOV)       := EX.o = $(MARKOV.o)
316 $(MARKOV.ln)    := EXAMPLE = -DMARKOV
317 $(MARKOV.ln)    := EX.c = $(MARKOV.c)
318
319 $(TIMER)        := EXAMPLE = -DTIMER
320 $(SEND)         := EXAMPLE = -DSEND
321
322 #  Uncomment the following lines on systems that do not support
323 #  conditional macros.
324 #
325 #EXAMPLE        = -DDIAG
326 #EX.c           = $(DIAG.c)
327 #EX.o           = $(DIAG.o)
328
329
330
331 #  Everything.
332
333 all : $(DIAG) $(DIAG.ln) $(MARKOV) $(MARKOV.ln) $(TIMER) $(SEND) $(ARFILE)
334
335
336
337 #  Common production rules.
338
339 .c.o :
340         $(CC) $(EXAMPLE) $(CFLAGS) $(SETUP) -c $< 2| $(TEE) $(ERRFILE)
341
342 $(DIAG.ln) $(MARKOV.ln) :
343         $(LINT) $(EXAMPLE) $(LFLAGS) $(SETUP) $(EX.c) $(COM.c) > $@ 2>&1
344
345
346
347 #  Common object file dependencies.
348
```

```
349 aq.o : markov.h setup.h debug.h matrix.h go.h aq.h time.h
350 boot.o : markov.h setup.h matrix.h debug.h go.h
351 go.o : markov.h setup.h debug.h matrix.h go.h aq.h qr.h time.h
352 matrix.o : markov.h setup.h debug.h matrix.h time.h
353 qr.o : markov.h setup.h debug.h matrix.h qr.h time.h
354 srrit.o : markov.h setup.h debug.h matrix.h
355
356
357
358 #  Computational examples.
359
360 $(DIAG) $(MARKOV) : $(DOM.a) $(COM.o)
361         $(LD) $(EX.o) $(COM.o) $(DOM.a) $(PRO.a) $(CLIBS) $(LDFLAGS) -o $@
362         @$(SIZE) $@
363
364 $(DIAG) : $(DIAG.o)
365 $(DIAG.o) : diag.h setup.h matrix.h aq.h debug.h time.h
366
367 $(MARKOV) : $(MARKOV.o)
368 $(MARKOV.o) : markov.h setup.h matrix.h aq.h debug.h
369
370
371
372 #  Timers.
373
374 $(TIMER) : $(DOM.a) $(TIMER.o)
375         $(CC) $(TIMER.o) $(DOM.a) $(PRO.a) $(CLIBS) $(LDFLAGS) -o $@
376         @$(SIZE) $@
377
378 $(TIMER.o) : time.h
379
380 $(SEND) : $(DOM.a) $(SEND.o)
381         $(CC) $(SEND.o) $(DOM.a) $(PRO.a) $(CLIBS) $(LDFLAGS) -o $@
382         @$(SIZE) $@
383
384 $(SEND.o) : time.h
385
386
387
388 #  DOMINO libraries.
389
390 $(DOM.a) :
391         cd $(DOMDIR) ; make
392
393 $(PRO.a) : $(DOM.a)
394
395
396
397 #  Lint checking.
398
399 sure check lint : $(MARKOV.ln) $(DIAG.ln)
400
401
402
403 #  Print listings.
404
405 listings $(PRFILE) :
406         for file in Makefile *.h *.c ; \
407         do \
408                 cat -n $$file | pr -f -h "   Thesis file:  $$file   " ; \
409         done > $(PRFILE)
410
411
412
413 #  Tape archive.
414
415 $(ARFILE) $(AR) :
416         $(AR) $(ARFLAGS) *
417
418
419
420 #  Cleanup.
421
422 clean : FORCE
423         rm -f *.o *.bak core ,* $(ERRFILE) $(ARFILE) $(PRFILE)
424
425 clobber : clean
426         rm -f $(DIAG) $(DIAG.ln) $(MARKOV) $(MARKOV.ln) $(TIMER) $(SEND)
427
428 realclean : clobber
429         cd $(DOMDIR) ; make realclean
430
431
432
433 #  Force a certain operation to be performed.
434
435 FORCE:
```

```
/*
 *  AQ Node Definitions
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:21:44
 *
 *  VERSION:    @(#)aq.h 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains definitions for the structures used by
 *      the nodes that circularly calculate the product of matrices
 *      A and Q.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */



#ifndef _SETUP_H
#include "setup.h"
#endif


/*
*  AQSTACKSIZE is the size of the DOMINO stack for an AQ node.
*/

#define AQSTACKSIZE     40000


/*
*  AQAUXSIZE is the size of the structure used for the auxiliary storage
*  for the AQ node in integers (for DOMINO).
*/

#define AQAUXSIZE       (sizeof (struct aqaux) / sizeof (int))


/*
*  The aqinfo structure is used to convey processing control information
*  to the AQ node.  Its fields represent the lower and upper bounds of
*  the columns to use in the multiplication as well as a flag to indicate
*  whether or not the T matrix should be calculated in parallel.
*/

struct aqinfo {
        int lower;              /* Lower bound of multiply */
        int upper;              /* Upper bound of multiply */
        BOOLEAN calculate_T;    /* Flag for calculation of T */
        BLOCK *bq;              /* Pointer to block array */
};


/*
*  Following is the structure used for the DOMINO auxiliary storage
*  for an AQ node.
*/

struct aqaux {
        int aqnum;                              /* AQ node number */
        int aqrowindex;                         /* Row index of node */
        int aqrowdim;                           /* Row dimension of block */
        int aqcounter [NAMESIZE (NAMELENGTH)];  /* Counter-clockwise node ID */
        int aqclock [NAMESIZE (NAMELENGTH)];    /* Clockwise node ID */
        struct nodeid *aqcontrol;               /* Control node ID */
        BOOLEAN aqmflag;                        /* Master flag */
#ifdef MARKOV
        struct placeinfo aqplaceinfo [BLOCKDIM]; /* Place information */
#endif /* MARKOV */
};


/*
 *  Global data.
 */

extern int firstaq [];


/*
```



```
 *  Function prototypes.
 */

void aqmult ( /* struct node *, int * */ );
void atq ( /* MAT, MAT, MAT, int, int, BOOLEAN */ );
void aqinit ( /* struct aqaux * */ );


#define _AQAUX_H
```

```
/*
 *  Debugging Macro Definitions
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:22:15
 *
 *  VERSION:    @(#)debug.h 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains definitions for debugging facilities used
 *      by the various routines in the thesis.  Specifically, a debug
 *      level is declared for the entire program and individual debug
 *      macro calls may either be activated or deactivated by specifying
 *      a debug level in the call.  If the level specified in the macro
 *      expansion is greater than or equal to the declared program-wide
 *      debug level, then a debug message will be issued.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */



#undef   NULL                   /* Redeclared by DOMINO header files */
#include <stdio.h>


#define DBOFF           0
#define DBNICE          1
#define DBMUST          2

#define DEBUG           DBMUST


/*
 *  Macro for machine-independent flushing of output.
 */

#if VAX
#       define flush()                  fflush (stdout)
#else /* VAX */
#       if MCMOB
#               define flush()          serflush ()
#       else /* MCMOB */
#               define flush()
#       endif /* MCMOB */
#endif /* VAX */


/*
 *  Macros to indicate entry and exit of routines, showing file name and
 *  line number.
 */

#define ENTER(proc, dblevel) \
        { \
                if (dblevel >= DEBUG) { \
                        (void) printf ("%15s(%3d):  Entering routine %s\n", \
                                __FILE__, __LINE__, proc); \
                        flush (); \
                } \
        }

#define LEAVE(proc, dblevel) \
        { \
                if (dblevel >= DEBUG) { \
                        (void) printf ("%15s(%3d):  Leaving routine  %s\n", \
                                __FILE__, __LINE__, proc); \
                        flush (); \
                } \
        }

#define DBMSG(msg, dblevel) \
        { \
                if (dblevel >= DEBUG) { \
                        (void) printf ("%15s(%3d):  %s\n", \
                                __FILE__, __LINE__, msg); \
                        flush (); \
                } \
        }

```



```
#define _DEBUG_H
```

```
/*
 *  Random Diagonal Matrix Definitions
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:22:08
 *
 *  VERSION:    @(#)diag.h 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains definitions for a matrix example
 *                        T
 *      of the form D + uu  where D is a random diagonal
 *      matrix and u is a random vector.  Thus a matrix element is
 *      defined by
 *
 *                      u u  + delta  d
 *                       i j        ij i
 *
 *      where delta(i,j) is the Kronecker delta function.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */

/*
 *  DEGREEOFA is the degree of the matrix A.
 */

#define DEGREEOFA       100


/*
 *  Function prototypes.
 */

void aqmultiply ( /* struct aqaux *, int, BLOCK *, BLOCK *, int, int */ );


#define _DIAG_H
```

```
/*
 *  Markov Chain Example Definitions
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:21:53
 *
 *  VERSION:    @(#)markov.h 4.1
 *
 *  DESCRIPTION:
 *
 *     This file contains definitions for the Markov chain example
 *     used by this program.
 *
 *     The contents of this file form part of an appendix to the thesis,
 *     "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *     for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *     Matrices", by John R. Meyer and submitted to the faculty of the
 *     Graduate School of the University of Maryland in partial fulfillment
 *     of the requirements for the degree of master of science.
 */


/*
 *  The size of the grid used in the Markov chain example.
 */

#define GRIDSIZE        10


/*
 *  DEGREEOFA is the degree of the transition matrix A.
 */

#define DEGREEOFA       (GRIDSIZE * (GRIDSIZE + 1) / 2)


/*
 *  Following are the directions used for the AQ node as well as the
 *  structure needed to keep track of a vertex's neighbors.
 *
 *  NOTE:       Routine markov.aqmultiply requires that NORTH be
 *              the first identifier in the direction type and
 *              that WEST be the last.
 */

typedef enum {
        NORTH,
        SOUTH,
        EAST,
        WEST
} DIRECTION;

struct placeinfo {
        int vert;               /* Vertical coordinate */
        int horiz;              /* Horizontal coordinate */
        int neighbors [4];      /* Neighbors of point on grid */
};


/*
 *  Function prototypes.
 */

void aqmultiply ( /* struct aqaux *, int, BLOCK *, BLOCK *, int, int */ );


#define _MARKOV_H
```

```c
/*
 *  Matrix, Vector, and Block Definitions
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:22:18
 *
 *  VERSION:    @(#)matrix.h 4.1
 *
 *  DESCRIPTION:
 *
 *     This file contains definitions for the matrix and vector data types
 *     used by the routines in this package.
 *
 *     This file requires that the preprocessor constant DEGREEOFA be
 *     defined, presumably by the header file containing the definitions
 *     for the particular computational example being used.
 *
 *     The contents of this file form part of an appendix to the thesis,
 *     "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *     for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *     Matrices", by John R. Meyer and submitted to the faculty of the
 *     Graduate School of the University of Maryland in partial fulfillment
 *     of the requirements for the degree of master of science.
 */



#ifndef _SETUP_H
#include "setup.h"
#endif


/*
 *  COLUMNSOFQ is the number of columns in the Q matrix, that is, the
 *  dimension of the invariant subspace used.
 */

#ifndef COLUMNSOFQ
#define COLUMNSOFQ      4
#endif


/*
 *  NUMVECTORS is the number of eigenvectors of Q to calculate.
 */

#ifndef NUMVECTORS
#define NUMVECTORS      2
#endif


/*
 *  ITVECTORS is the number of vectors of Q to iterate with.
 */

#ifndef ITVECTORS
#define ITVECTORS       4
#endif


/*
 *  BLOCKDIM is the row dimension of each block of Q.
 */

#define BLOCKDIM        ((DEGREEOFA % NUMVIRPROCS == 0) \
                                ? DEGREEOFA / NUMVIRPROCS \
                                : DEGREEOFA / NUMVIRPROCS + 1)



/*
 *  Following are the data types used to represent the matrices and
 *  vectors and their block forms as well as macros to access their elements.
 */

#define VECX(vec,index)         ((vec)[(index)-1])
#define IVECX(vec,index)        ((vec)[(index)-1])
#define MATX(mat,row,col)       ((mat)[(row)-1][(col)-1])
#define BMATX(mat,row,col)      ((mat)[(row)-1][(col)-1])
#define SBMATX(mat,row)         ((mat)[(row)-1])

typedef float VEC [DEGREEOFA];
typedef int IVEC [DEGREEOFA];
typedef float COLMAT [DEGREEOFA] [COLUMNSOFQ];
typedef float MAT [COLUMNSOFQ] [COLUMNSOFQ];
typedef float BLOCKMAT
```



```c
        [BLOCKDIM > COLUMNSOFQ ? BLOCKDIM : COLUMNSOFQ] [COLUMNSOFQ];
typedef float SBMAT [COLUMNSOFQ * (COLUMNSOFQ + 1) / 2];


/*
 *  BLOCK is the structure used to represent a block of Q.
 */

#define BLOCKSIZE               (sizeof (BLOCK) / sizeof (int))
#define BLKREF(block,row,col)   MATX((block).blkmat,row,col)

typedef struct {
        int begrow;             /* First row index of block */
        int nbrow;              /* Number of rows in block */
        BLOCKMAT blkmat;        /* Storage for the block */
} BLOCK;


/*
 *  SYMBLOCK is the structure used to represent a block of a symmetric matrix.
 */

#define SBSIZE                  (sizeof (SYMBLOCK) / sizeof (int))
#define SBLKREF(block,index)    SBMATX((block).blkmat,index)

typedef struct {
        int begrow;             /* First row index of block */
        int nbrow;              /* Number of rows in block */
        SBMAT blkmat;           /* Storage for the symmetric block */
} SYMBLOCK;




/*
 *  Type definitions for the eigenvalue types.
 *
 *  NOTE:  Each of the following enumerated constants must have the
 *         indicated integer values since srrit.group depends on them.
 */


#define EVECX(vec,index)        ((vec)[(index)-1])

typedef enum {
        REAL_EIG,               /* Real eigenvalue */
        COMPLEX_POS,            /* Complex eigenvalue positive imaginary */
        COMPLEX_NEG,            /* Complex eigenvalue negative imaginary */
        NO_SUCCESS = -1         /* No convergence */
} EIG_TYPE, TYPEVEC [DEGREEOFA];



/*
 *  Row and column indexes.
 */


#define ROWINDEX(aqnum) \
        ((DEGREEOFA % NUMVIRPROCS == 0) \
                ? ((aqnum) * BLOCKDIM + 1) \
                : ((aqnum) * BLOCKDIM + 1 + \
                        (((aqnum) <= DEGREEOFA % NUMVIRPROCS) \
                                ? 0 \
                                : DEGREEOFA % NUMVIRPROCS - (aqnum))))

#define ROWDIM(aqnum) \
        (DEGREEOFA / NUMVIRPROCS + \
                (((aqnum) < DEGREEOFA % NUMVIRPROCS) ? 1 : 0))



/*
 *  Type definitions for the action to be taken concerning the
 *  initialization of the matrix Q.
 */

typedef enum {
        RANDOM,                 /* Generate a random matrix */
        ORTRANDOM,              /* Generate a random orthonormal matrix */
        QCOLUMNS,               /* Leave the columns of Q as is */
        ORTQCOLUMNS             /* Orthogonalize the existing columns of Q */
} QINITIAL;


/*
 *  Function prototypes.
```

```
175  */
176
177 void initblk ( /* BLOCK *, int, int */ );
178 void sumblk ( /* BLOCK *, BLOCK * */ );
179 void multblk ( /* BLOCK *, BLOCK * */ );
180 void qttq ( /* BLOCK *, SYMBLOCK * */ );
181 void mat2blk ( /* MAT, BLOCK *, int, int */ );
182 void blk2mat ( /* BLOCK *, MAT */ );
183 void sblksum ( /* SYMBLOCK *, SYMBLOCK * */ );
184 void blkprt ( /* BLOCK *, char * */ );
185 float dot ( /* int, float [], float [] */ );
186 void cholesky ( /* SBMAT, int int * */ );
187 void solve ( /* BLOCK *, SYMBLOCK *, BLOCK * */ );
188 void srrit ( /* MAT, MAT, void (*) (), int, int *, m, float, int,
189         QINITIAL, MAT, VEC, VEC, TYPEVEC, VEC, IVEC */ );
190
191 #define _MATRIX_H
```

```
/*
 *  QR Node Definitions
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:21:54
 *
 *  VERSION:    @(#)qr.h 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains definitions for the auxiliary storage structure
 *      used by the nodes which calculate the QR reorthogonalization.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */



#ifndef _SETUP_H
#include "setup.h"
#endif


/*
 *  BINSUM determines if a binary summation is to be performed during the
 *  QR reorthogonalization.
 */

#ifndef BINSUM
#define BINSUM          TRUE
#endif


/*
 *  QRAUXSIZE is the size of the auxiliary storage of a QR node.
 *  QRSTACKSIZE is the size of the stack of a QR node.
 */

#define QRAUXSIZE       ((sizeof (struct qraux) / sizeof (int)))
#define QRSTACKSIZE     60000


/*
 *  Following is the structure used for the DOMINO auxiliary storage for
 *  a QR node.
 */

struct qraux {
        int qrnum;                              /* QR node number */
        int qrcounter [NAMESIZE (NAMELENGTH)];  /* Counter-clockwise node ID */
        int qrclock [NAMESIZE (NAMELENGTH)];    /* Clockwise node ID */
        struct nodeid *qrcontrol;               /* Control node ID */
        BOOLEAN qrmflag;                        /* Master flag */
#if BINSUM
        int qrnadd;                             /* Number of additions */
        int qrnpass;                            /* Number of passes */
#endif /* BINSUM */
};


/*
 *  Global data.
 */

extern int firstqr [];


/*
 *  Function prototypes.
 */

void qr ( /* struct node *, int * */ );

#if BINSUM
void getends ( /* int, int *, int * */ );
int numadd ( /* int, int, int */ );
int numpass ( /* int, int, int */ );
#endif /* BINSUM */


#define _QRAUX_H
```

```c
/*
 *  Machine and Processor Ring Setup Definitions
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:22:21
 *
 *  VERSION:    @(#)setup.h 4.1
 *
 *  DESCRIPTION:
 *
 *     This file contains constants used for the set-up structure of
 *     the computational network.
 *
 *     The contents of this file form part of an appendix to the thesis,
 *     "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *     for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *     Matrices", by John R. Meyer and submitted to the faculty of the
 *     Graduate School of the University of Maryland in partial fulfillment
 *     of the requirements for the degree of master of science.
 */


/*
 *  Read in the appropriate header file for the computational example.
 */

#ifdef DIAG
#include "diag.h"               /* Random diagonal test example */
#endif

#ifdef MARKOV
#include "markov.h"             /* Markov chain example */
#endif


/*
 *  Data types.
 */

typedef unsigned int BOOLEAN;

#ifndef TRUE
#define TRUE            1              /* Logical true value */
#define FALSE           0              /* Logical false value */
#endif


/*
 *  Processor-ring definitions.
 */

#ifndef NUMVIRPROCS
#define NUMVIRPROCS     5       /* The number of "virtual" processors,
                                 * that is, the number simulated on
                                 * single-processor machines and the
                                 * actual number on parallel machines */
#endif

#ifndef MAXPROCS
#define MAXPROCS        512     /* The maximum number of processors
                                 * that this program can handle, which
                                 * must be an integral power of 2 */
#endif

#ifndef VERBOSE
#define VERBOSE         TRUE    /* Print out program setup statistics */
#endif

#ifndef EPSILON
#define EPSILON         1.0e-05 /* Error tolerance */
#endif

#ifndef MAXIT
#define MAXIT           1000    /* Maximum number of iterations to perform */
#endif


/*
 *  Definitions for different versions of DOMINO.
 */

#if MCMOB
#define NQENTS          nqents
#define NLENTS          nlents
#define STOREX          storex
#define QPOOLP          qpoolp
```



```c
#define LPOOLP          lpoolp
#define STORE           store
#define SELFADDR        selfaddr
#define NODE            node
#define NNODES          nnodes
#endif /* MCMOB */


/*
 *  Constants and macros describing special nodes in the ring.
 */

#define FIRST                   0
#define LAST                    (NUMVIRPROCS - 1)
#define MASTER                  FIRST

#define ISFIRST(node)           ((node) == FIRST)
#define ISLAST(node)            ((node) == LAST)
#define ISMASTER(node)          ((node) == MASTER)


/*
 *  Macro for the size of a DOMINO name.
 */

#define NAMESIZE(length)        (2 * (length) + 3)


/*
 *  Following are indices into the node IDs that store the node type
 *  (AQ or QR) and the processor number
 */

#define NAMELENGTH              2
#define PROCESSOR               0
#define NODETYPE                1


/*
 *  Definitions for the different node types.
 */

typedef enum {
        GOTYPE,
        AQTYPE,
        QRTYPE
} TYPEOFNODE;


/*
 *  Function prototypes.
 */

void srrerror ( /* char * */ );


#define _SETUP_H
```

```c
/*
 *  Timing Macro Definitions
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:22:02
 *
 *  VERSION:    @(#)time.h 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains definitions for the clock timer functions
 *      necessary for the guaging of the efficiency of the parallel
 *      network.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */


long int clock ();


#if SINGLE_PROCESSOR

#define TOMICROSECS(ticks)      (ticks)
#define TIMERSTART(print, msg, start)
#define TIMERPAUSE(print, msg, start, total)
#define TIMEREND(print, msg, start, end)
#define TIMERPRINT(print, msg, total)

#else /* SINGLE_PROCESSOR */

#define TOMICROSECS(ticks)      ((ticks) * 0.407)

#define TIMERSTART(print, msg, start) \
        { \
                if (print) { \
                        (void) printf ("%15s(%3d):  Starting timer: %s\n", \
                                __FILE__, __LINE__, msg); \
                        flush (); \
                } \
                start = clock (); \
        }

#define TIMERPAUSE(print, msg, start, total) \
        { \
                total += clock () - start; \
                if (print) { \
                        (void) printf ("%15s(%3d):  Pausing timer: %s\n", \
                                __FILE__, __LINE__, msg); \
                        flush (); \
                } \
        }

#define TIMEREND(print, msg, start, total) \
        { \
                total = clock () - start; \
                if (print) { \
                        (void) printf ("%15s(%3d):  Ending timer: %s\n", \
                                __FILE__, __LINE__, msg); \
                        flush (); \
                } \
        }

#define TIMERPRINT(print, msg, total) \
        { \
                if (print) { \
                        (void) printf ("%15s(%3d):  %s: %.4f secs\n", \
                                __FILE__, __LINE__, msg, \
                                TOMICROSECS (total) * 1.0e-6); \
                        flush (); \
                } \
        }

#endif /* SINGLE_PROCESSOR */


#define _TIMER_H
```

```
/*
 *  AQ Node Program
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:21:56
 *
 *  VERSION:    @(#)aq.c 4.1
 *
 *  DESCRIPTION:
 *
 *     This file contains the code for the node program which calculates
 *     the product of matrices A and Q in a circular fashion.
 *
 *     The major steps in this calculation are:
 *
 *             - Collect the leftmost and rightmost endpoints of the
 *               interval to be used in the multiplication and send
 *               them to the right neighbor.
 *
 *             - Pass the remaining blocks of Q around the ring.
 *
 *             - Collect this node's block of Q.
 *
 *             - Perform a circular inner product of A and the blocks
 *               of Q, rotating the blocks of Q around the ring of
 *               processors.
 *
 *             - Send this node's accumulated sum to the left neighbor.
 *
 *             - Pass the remaining sums from the right neighbors to
 *               the left.
 *
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */



/*
 *  Binary SCCS identification.
 */

#ifndef lint
static char sccsid [] = "aq.c 4.1";
#endif


/*
 *  Header file inclusions.
 */


#include "domparam.h"
#include "domstruct.h"
#include "domdec.h"

#include "setup.h"
#include "debug.h"
#include "matrix.h"
#include "go.h"
#include "aq.h"
#include "time.h"



/* ARGSUSED */


void aqmult (nd, sysp)
/*
 *  Procedure aqmult is the node program for the AQ node.
 *
 *  The parameters represent:
 *
 *      nd              A pointer to the AQ node structure
 *      sysp            A pointer to DOMINO system variables
 */

struct node *nd;
int *sysp;

```



```
{
        auto struct aqaux *ap = (struct aqaux *) nd->auxp;
                                        /* Auxiliary storage pointer */
        auto BLOCK q;                   /* Block of Q matrix */
        auto BLOCK sum;                 /* Block containing running sum */
        auto BLOCK temp;                /* Temporary block storage */
        auto struct aqinfo info;        /* Execution information */
        auto int npass;                 /* Number of times to pass blocks */
        auto long int sclk, tclk;       /* Clockwise transmission time */
        auto long int srot, trot;       /* Rotation transmission time */
        auto long int sclc, tclc;       /* Calculation time */
        auto long int scnt, tcnt;       /* Counter-clockwise time */
        auto long int stot, ttot;       /* Total time */



        ENTER ("aqmult", DBOFF);


        /*
         * Collect the directions for this iteration.
         */

        request (ap->aqmflag ? ap->aqcontrol : (struct nodeid *) ap->aqcounter,
                (int *) &info);
        pause ();
        if (!ISLAST (ap->aqnum))
                sendn ((struct nodeid *) ap->aqclock,
                        sizeof info / sizeof (int), (int *) &info);

        TIMERSTART (FALSE, "starting total timer", stot);


        /*
         * Pass the neighboring blocks of Q clockwise amd collect our
         * own block.
         */

        TIMERSTART (FALSE, "starting AQ clockwise timer", sclk);
        for (npass = LAST - ap->aqnum; npass > 0; npass--) {
                if (ap->aqmflag)
                        sendn ((struct nodeid *) ap->aqclock, BLOCKSIZE,
                                (int *) (info.bq + npass));
                else {
                        request ((struct nodeid *) ap->aqcounter,
                                (int *) &temp);
                        pause ();
                        sendn ((struct nodeid *) ap->aqclock, BLOCKSIZE,
                                (int *) &temp);
                }
        }
        TIMEREND (FALSE, "stopping AQ clockwise timer", sclk, tclk);


        /*
         *  Collect our own block of Q.
         */

        if (ap->aqmflag)
                q = info.bq[0];
        else {
                request ((struct nodeid *) ap->aqcounter, (int *) &q);
                pause ();
        }


        /*
         *  Circularly perform an inner product of A and the blocks of Q,
         *  rotating the blocks of Q around the ring of processors.
         */

        initblk (&sum, q.begrow, BLOCKDIM);

        tclc = 0;
        TIMERSTART (FALSE, "starting AQ rotation timer", srot);
        for (npass = NUMVIRPROCS; npass > 0; npass--) {
                TIMERSTART (FALSE, "continue AQ calc timer", sclc);
                aqmultiply (ap, ap->aqrowdim, &q, &sum, info.lower, info.upper);
                TIMERPAUSE (FALSE, "stopping AQ calc timer", sclc, tclc);
                sendn ((struct nodeid *) ap->aqclock, BLOCKSIZE, (int *) &q);
                request ((struct nodeid *) ap->aqcounter, (int *) &q);
                pause ();
        }
        TIMEREND (FALSE, "ending AQ rotation timer", srot, trot);


        /*
```

```
175          *  Send our own block followed by the remaining blocks around
176          *  the ring.
177          */
178
179         if (ap->aqmflag)
180                 info.bq[0] = sum;
181         else
182                 sendn ((struct nodeid *) ap->aqcounter, BLOCKSIZE,
183                         (int *) &sum);
184
185         TIMERSTART (FALSE, "starting AQ collection timer", scnt);
186         for (npass = 1; npass <= LAST - ap->aqnum; npass++) {
187                 request ((struct nodeid *) ap->aqclock, (int *) &temp);
188                 pause ();
189                 if (ap->aqmflag)
190                         info.bq[npass] = temp;
191                 else
192                         sendn ((struct nodeid *) ap->aqcounter, BLOCKSIZE,
193                                 (int *) &temp);
194         }
195         TIMEREND (FALSE, "ending AQ collection timer", scnt, tcnt);
196
197         TIMEREND (FALSE, "ending total timer", stot, ttot);
198
199
200         /*
201          *  Signal the go-node that we are finished with A * Q.
202          */
203
204         if (ap->aqmflag)
205                 sendn ((struct nodeid *) ap->aqcontrol, 0, (int *) NULL);
206
207
208         /*
209          *  Calculate T, if necessary.
210          */
211
212         if (info.calculate_T) {
213                 multblk (&q, &sum);
214                 if (!ISLAST (ap->aqnum)) {
215                         request ((struct nodeid *) ap->aqclock, (int *) &q);
216                         pause ();
217                         sumblk (&q, &sum);
218                 }
219                 if (ap->aqmflag)
220                         sendn (ap->aqcontrol, BLOCKSIZE, (int *) &sum);
221                 else
222                         sendn ((struct nodeid *) ap->aqcounter, BLOCKSIZE,
223                                 (int *) &sum);
224         }
225
226
227         /*
228          *  Print out the timing results.
229          */
230
231         TIMERPRINT (TRUE, "Total AQ clockwise circulation time", tclk);
232         TIMERPRINT (TRUE, "Total AQ calculation time", tclc);
233         TIMERPRINT (TRUE, "Total AQ rotation time", trot - tclc);
234         TIMERPRINT (TRUE, "Total AQ counter-clockwise transmission time", tcnt);
235         TIMERPRINT (TRUE, "Total AQ time", ttot);
236
237
238
239         LEAVE ("aqmult", DBOFF);
240 }
241
242
243
244
245 #define Q(i,j)          MATX(q,i,j)
246 #define AQ(i,j)         MATX(aq,i,j)
247 #define T(i,j)          MATX(t,i,j)
248
249
250
251 void atq (q, aq, t, lower, upper, calc_T)
252 /*
253  *  Procedure atq is the interface between the sequential and parallel
254  *  parts of this program, activating the AQ ring to calculate the product.
255  *
256  *  The parameters represent:
257  *
258  *      q               The entire Q matrix which will be broken into
259  *                      blocks and sent around the ring.
260  *      aq              The entire product AxQ whose blocks are calculated
261  *                      by the ring.
```



```
262  *      lower           The lower (leftmost) index of the Q to be used in
263  *                      the multiplication.
264  *      upper           The upper (rightmost) index of Q to be used in the
265  *                      multiplication.
266  *      calc_T          A flag indicating whether or not T is to be calculated.
267  */
268
269 BOOLEAN calc_T;
270 int lower, upper;
271 MAT q, aq, t;
272
273 {
274         auto int blkidx;
275         auto BLOCK bqt;
276         auto struct aqinfo info;
277         static BLOCK bq [NUMVIRPROCS];
278
279
280
281         ENTER ("atq", DBOFF);
282
283
284         /*
285          *  Break up Q into blocks.
286          */
287
288         for (blkidx = 0; blkidx < NUMVIRPROCS; blkidx++)
289                 mat2blk (q, bq + blkidx, ROWINDEX (blkidx), ROWDIM (blkidx));
290
291
292         /*
293          *  Collect the relevant control information together and
294          *  tell the master node to begin processing.
295          */
296
297         info.lower = lower;
298         info.upper = upper;
299         info.calculate_T = calc_T;
300         info.bq = bq;
301         sendn ((struct nodeid *) firstaq, sizeof info / sizeof (int),
302                 (int *) &info);
303
304
305         /*
306          *  Collect the blocks of the calculated product and put them back
307          *  together in AQ.  Also, if calculation of T is requested, place
308          *  the result in T.
309          */
310
311         request ((struct nodeid *) firstaq, (int *) NULL);
312         pause ();
313         for (blkidx = 0; blkidx < NUMVIRPROCS; blkidx++)
314                 blk2mat (bq + blkidx, aq);
315
316         if (calc_T) {
317                 request ((struct nodeid *) firstaq, (int *) &bqt);
318                 pause ();
319                 blk2mat (&bqt, t);
320         }
321
322
323
324         LEAVE ("atq", DBOFF);
325 }
326
327
328 #undef  Q
329 #undef  AQ
330 #undef  T
```

```
/*
 *  Boot Node Program
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:21:40
 *
 *  VERSION:    @(#)boot.c 4.1
 *
 *  DESCRIPTION:
 *
 *     This file contains code for the boot node, which initializes the
 *     queue and list pools as well as creates the go nodes for this processor.
 *
 *     The contents of this file form part of an appendix to the thesis,
 *     "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *     for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *     Matrices", by John R. Meyer and submitted to the faculty of the
 *     Graduate School of the University of Maryland in partial fulfillment
 *     of the requirements for the degree of master of science.
 */



/*
 *  Binary SCCS identification.
 */

#ifndef lint
static char sccsid [] = "boot.c 4.1";
#endif



/*
 *  Header file inclusions.
 */


#include "domparam.h"
#include "domstruct.h"
#include "domdec.h"

#include "setup.h"
#include "matrix.h"
#include "debug.h"
#include "go.h"



/*
 *  DOMINO-related constants.
 */

#define NUMQELMNTS      2000    /* Number of queue elements */
#define NUMLISTELMNTS   4000    /* Number of list elements */





#if  MCMOB

/*
 *  The processor table is an array containing the actual frob numbers
 *  corresponding to the SELFADDR value for the processors on the
 *  machine.  For example, the index of the element containing 1 is
 *  the SELFADDR for frob 1.
 */

unsigned short int proctabl [] = {
        0x00ff, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0008,
        0x000a, 0x000b, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
};

#endif /* MCMOB */



/* ARGSUSED */


void boot (nd, sysp)
struct node *nd;                        /* Self node structure */
int *sysp;                              /* System pointer */

{
        ENTER ("boot", DBOFF);
```



```


#if MCMOB

        /*
         * Initialize the queue pool.
         */

        {
                auto int element;
                auto struct qentry *qstart;

                NQENTS = NUMQELMNTS;
                if (STOREX + NQENTS * QENTSIZE >= STORSIZE)
                        srerror ("\nboot:  too many queue entries.\n");
                QPOOLP = qstart = (struct qentry *) STORE + STOREX;
                for (element = 1; element <= NQENTS; element++)
                        if (element != NQENTS) {
                                QPOOLP->nextent = QPOOLP + 1;
                                QPOOLP++;
                        } else
                                QPOOLP->nextent = NULL;
                QPOOLP = qstart;
                STOREX += NQENTS * QENTSIZE;
        }


        /*
         * Initialize the list pool.
         */

        {
                auto int element;
                auto struct listentry *lstart;

                NLENTS = NUMLISTELMNTS;
                if (STOREX + NLENTS * LENTSIZE >= STORSIZE)
                        srerror ("\nboot:  too many list entries.\n");
                LPOOLP = lstart = (struct listentry *) STORE + STOREX;
                for (element = 1; element <= NLENTS; element++)
                        if (element != NLENTS) {
                                LPOOLP->nextent = LPOOLP + 1;
                                LPOOLP++;
                        } else
                                LPOOLP->nextent = NULL;
                LPOOLP = lstart;
                STOREX += NLENTS * LENTSIZE;
        }

#else  /* MCMOB */

        if (!makepools (NUMQELMNTS, NUMLISTELMNTS))
                error (71);

#endif /* MCMOB */



        /*
         * Make the go nodes.
         */

        {
                static struct goaux *goap;
                static name [NAMELENGTH] = { 0 };

#if SINGLE_PROCESSOR

                static int virproc, procnum;

                GOAUXSIZE = sizeof (struct goaux) / sizeof (int) + 1;
                GOSTKSIZE = GOSTACKSIZE;
                name[NODETYPE] = (int) GOTYPE;

                for (virproc = 0; virproc < NUMVIRPROCS; virproc++) {
                        procnum = SELFADDR + virproc - 1;
                        name[PROCESSOR] = procnum;
                        makenode (USER, NAMELENGTH, name, go, GOAUXSIZE + 1,
                                GOSTACKSIZE);
                        goap = (struct goaux *) NODE[NNODES - 1]->auxp;
                        goap->gonum = procnum;
                        goap->gomflag = ISMASTER (procnum) ? TRUE : FALSE;
                        makeid (SELFADDR, 0, NAMELENGTH, name,
                                (struct nodeid *) goap->goid);
                }

#else /* SINGLE_PROCESSOR */
```

```
175
176 #       if !MCMOB
177                 GOAUXSIZE = sizeof (struct goaux) / sizeof (int);
178                 GOSTKSIZE = GOSTACKSIZE;
179 #       endif /* !MCMOB */
180
181                 name[NODETYPE] = (int) GOTYPE;
182                 name[PROCESSOR] = SELFADDR - 1;
183                 makenode (USER, NAMELENGTH, name, go, GOAUXSIZE, GOSTACKSIZE);
184                 goap = (struct goaux *) NODE[NNODES - 1]->auxp;
185                 goap->gonum = SELFADDR - 1;
186                 goap->gomflag = (SELFADDR == 1) ? TRUE : FALSE;
187                 makeid (SELFADDR, 0, NAMELENGTH, name,
188                         (struct nodeid *) goap->goid);
189
190 #endif /* SINGLE_PROCESSOR */
191
192         }
193
194         flush ();
195
196         LEAVE ("boot", DBOFF);
197         finis ();
198 }
199
200
201
202 void srrerror (message)
203 /*
204  *  Procedure srrerror prints the given error message on the standard output
205  *  and exits with an error status.
206  *
207  *  The parameter represents:
208  *
209  *      message         A message to be displayed on standard output.
210  */
211
212 char *message;
213
214 {
215         (void) printf ("%s\n", message);
216         flush ();
217         finis ();
218 }
```

```
/*
 *  Diagonal Matrix
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:22:07
 *
 *  VERSION:    @(#)diag.c 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains code for a matrix example
 *                                  T
 *      of the form D + uu  where D is a random diagonal
 *      matrix and u is a random vector.  Thus a matrix element is
 *      defined by
 *
 *                      u u  + delta  d
 *                       i j        ij i
 *
 *      where delta(i,j) is the Kronecker delta function.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */


/*
 *  Binary SCCS identification.
 */

#ifndef lint
static char sccsid [] = "diag.c 4.1";
#endif


/*
 *  Header file inclusions.
 */

#include "setup.h"
#include "matrix.h"
#include "aq.h"
#include "debug.h"
#include "time.h"



/*
 *  Local function prototypes.
 */

static float drand ( /* void */ );



/*
 *  Local variables.
 */

static float diag [DEGREEOFA];          /* Random diagonal matrix */
static float u [DEGREEOFA];             /* Random vector */



#define D(i)            diag[(i)-1]
#define U(i)            u[(i)-1]
#define SUM(i,j)        BLKREF(*sum,i,j)
#define Q(i,j)          BLKREF(*q,i,j)


/* ARGSUSED */

void aqmultiply (ap, rowdim, q, sum, lower, upper)
/*
 *  Procedure aqmultiply() provides the necessary floating point
 *  operations for this test example.
 *
 *  The parameters represent:
 *
 *      ap              A pointer to the auxiliary storage of the AQ node.
 *      rowdim          The row dimension of the block of Q.
 *      q               A block of the matrix Q that has just arrived at
```



```
 *                      this node to be "multiplied" by A.
 *      sum             Our own accumulation of a block of Q.
 *      lower           The lower (leftmost) index of the column of Q to
 *                      be multiplied.
 *      upper           The upper (rightmost) index of the column of Q to
 *                      be multiplied.
 */

int rowdim, lower, upper;
struct aqaux *ap;
BLOCK *q, *sum;

{
        auto int row, col;


        ENTER ("aqmultiply", DBOFF);


        for (row = 1; row <= rowdim; row++)
                for (col = 1; col <= COLUMNSOFQ; col++) {

                        register double acc = (double) 0.0;
                        register int index;

                        for (index = 1; index <= q->nbrow; index++) {
                                register float elem = U(index) * U(col) +
                                        (index == col) * D(index);
                                acc += (double) Q(row, index) * (double) elem;
                        }
                        SUM(row,col) += (float) acc;
                }


        LEAVE ("aqmultiply", DBOFF);
}


#undef D
#undef U
#undef SUM
#undef Q




#define D(i)            diag[(i)-1]
#define U(i)            u[(i)-1]


/* ARGSUSED */

void aqinit (aqptr)
/*
 *  Procedure aqinit places the random elements into the diagonal
 *  matrix and random vector.
 *
 *  The parameters represent:
 *
 *      aqptr   A pointer to the auxiliary storage of this AQ node.
 */

struct aqaux *aqptr;

{
        register int index;

        for (index = 1; index <= DEGREEOFA; index++) {
                D(index) = drand ();
                U(index) = drand ();
        }
}


#undef D
#undef U



static float drand ()
/*
 *  Author:    G. W. Stewart
 *  Modified:  John R. Meyer
 */

{
        static int seed = 69;
```

```
175
176         seed = (4621 * seed + 2113) % 10000;
177         return (float) seed / 10000.0;
178  }
```

```c
/*
 *  Go Node Program
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:21:46
 *
 *  VERSION:    @(#)go.c 4.1
 *
 *  DESCRIPTION:
 *
 *     This file contains code for the go-node.  Each virtual processor
 *     in this DOMINO program has one go-node, each of which creates an
 *     AQ node and a QR node.  If this node is on the master virtual
 *     processor, then it will also compute the sequential tasks of the
 *     thesis and report the results.
 *
 *     The contents of this file form part of an appendix to the thesis,
 *     "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *     for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *     Matrices", by John R. Meyer and submitted to the faculty of the
 *     Graduate School of the University of Maryland in partial fulfillment
 *     of the requirements for the degree of master of science.
 */



/*
 *  Binary SCCS identification.
 */

#ifndef lint
static char sccsid [] = "go.c 4.1";
#endif



/*
 *  Header file inclusions.
 */


#include "domparam.h"
#include "domstruct.h"
#include "domdec.h"

#include "setup.h"
#include "debug.h"
#include "matrix.h"
#include "go.h"
#include "aq.h"
#include "qr.h"
#include "time.h"



/*
 *  Master node IDs (used only by the control node).
 */

int firstaq [NAMESIZE (NAMELENGTH)];
int firstqr [NAMESIZE (NAMELENGTH)];




#if SINGLE_PROCESSOR
#define CNTPROC         SELFADDR       /* Address of counter-clockwise proc */
#define CLKPROC         SELFADDR       /* Address of clockwise processor */
#define INDEX           0
#else /* SINGLE_PROCESSOR */
#define CNTPROC         ((SELFADDR == 1) ? NUMVIRPROCS : SELFADDR - 1)
#define CLKPROC         ((SELFADDR == NUMVIRPROCS) ? 1 : SELFADDR + 1)
#define INDEX           (NNODES - 1)
#endif /* SINGLE_PROCESSOR */




#define ER(i)           VECX(er,i)
#define EI(i)           VECX(ei,i)


/* ARGSUSED */


void go (nd, sysp)
```



```c
struct node *nd;
int *sysp;

{
        auto struct goaux *ap = (struct goaux *) nd->auxp;


        ENTER ("go", DBOFF);



#if VERBOSE

        /*
         *  Print out vital statistics.
         */

        (void) printf ("\n*** SELFADDR %2d ***\n\n", SELFADDR);
        (void) printf ("Processors:       %d\n", NUMVIRPROCS);
        (void) printf ("Degree of A:      %d\t", DEGREEOFA);
        (void) printf ("Columns of Q:     %d\n", COLUMNSOFQ);
        (void) printf ("Block size:       %d ints\n", BLOCKSIZE);
        (void) printf ("Symmetric blk sz: %d ints\n", SBSIZE);
        (void) printf ("\n");
        flush ();

#endif /* VERBOSE */


        /*
         * Create the AQ node.
         */

        {
                auto struct aqaux *aqap;
                auto int aqname [NAMELENGTH];


                /*
                 *  Make the AQ node.
                 */

                aqname[NODETYPE] = (int) AQTYPE;
                aqname[PROCESSOR] = ap->gonum;
                makenode (USER, NAMELENGTH, aqname, aqmult,
                        AQAUXSIZE + 1, AQSTACKSIZE);

                /*
                 *  Initialize the new AQ node's auxiliary storage.
                 */

                aqap = (struct aqaux *) NODE[NNODES - 1]->auxp;
                aqap->aqnum = ap->gonum;
                aqap->aqrowindex = ROWINDEX (aqap->aqnum);
                aqap->aqrowdim = ROWDIM (aqap->aqnum);

#if VERBOSE
                (void) printf ("AQ row index:     %d\t", aqap->aqrowindex);
                (void) printf ("AQ row dimension: %d\n", aqap->aqrowdim);
                flush ();
#endif /* VERBOSE */

                /*
                 *  If this is the master node, make a node identifier
                 *  in the global variable firstaq.  This is the node
                 *  identifier of the first AQ node in the ring.
                 */

                aqap->aqmflag = ISMASTER (ap->gonum);
                if (aqap->aqmflag)
                        makeid (SELFADDR, INDEX, NAMELENGTH, aqname,
                                (struct nodeid *) firstaq);

                /*
                 *  Continue initializing the AQ node's auxiliary
                 *  storage, making the node identifiers for the left
                 *  and right neighbors of the node.
                 */

                aqname[PROCESSOR] = ISFIRST (ap->gonum) ? LAST : ap->gonum - 1;
                makeid (CNTPROC, INDEX, NAMELENGTH, aqname,
                        (struct nodeid *) aqap->aqcounter);
                aqname[PROCESSOR] = ISLAST (ap->gonum) ? FIRST : ap->gonum + 1;
                makeid (CLKPROC, INDEX, NAMELENGTH, aqname,
                        (struct nodeid *) aqap->aqclock);

                aqap->aqcontrol = (struct nodeid *) ap->goid;
```

```
175
176                 /*
177                  *  Call the computational example-dependent initialization
178                  *  routine for any other initialization that may need to
179                  *  take place.
180                  */
181
182                 aqinit (aqap);
183         }
184
185
186
187         /*
188          * Create the reorthogonalization node.
189          */
190
191         {
192                 auto struct qraux *qrap;
193                 auto int qrname [NAMELENGTH];
194 #if BINSUM
195                 auto int left, right;
196 #endif
197
198                 /*
199                  *  Make the QR node.
200                  */
201
202                 qrname[NODETYPE] = (int) QRTYPE;
203                 qrname[PROCESSOR] = ap->gonum;
204                 makenode (USER, NAMELENGTH, qrname, qr, QRAUXSIZE + 1,
205                         QRSTACKSIZE);
206
207                 /*
208                  *  Initialize the new QR node's auxiliary storage.
209                  */
210
211                 qrap = (struct qraux *) NODE[NNODES - 1]->auxp;
212                 qrap->qrnum = ap->gonum;
213 #if BINSUM
214                 getends (qrap->qrnum, &left, &right);
215                 qrap->qrnadd = numadd (qrap->qrnum, left, right);
216                 qrap->qrnpass = numpass (qrap->qrnum, left, right);
217 #endif /* BINSUM */
218
219 #if VERBOSE
220 #       if BINSUM
221                 (void) printf ("QR left end:      %d\t", left);
222                 (void) printf ("QR right end:     %d\n", right);
223                 (void) printf ("QR num additions: %d\t", qrap->qrnadd);
224                 (void) printf ("QR num passes:    %d\n", qrap->qrnpass);
225 #       endif /* BINSUM */
226                 flush ();
227 #endif /* VERBOSE */
228
229
230                 /*
231                  *  If this is the master node, make a node identifier
232                  *  in the global variable firstqr.  This is the node
233                  *  identifier of the first QR node in the ring.
234                  */
235
236                 qrap->qrmflag = ISMASTER (ap->gonum);
237                 if (qrap->qrmflag)
238                         makeid (SELFADDR, INDEX, NAMELENGTH, qrname,
239                                 (struct nodeid *) firstqr);
240
241                 /*
242                  *  Continue initializing the QR node's auxiliary
243                  *  storage, making the node identifiers for the left
244                  *  and right neighbors of the node.
245                  */
246
247                 qrname[PROCESSOR] = ISFIRST (ap->gonum) ? LAST : ap->gonum - 1;
248                 makeid (CNTPROC, INDEX, NAMELENGTH, qrname,
249                         (struct nodeid *) qrap->qrcounter);
250                 qrname[PROCESSOR] = ISLAST (ap->gonum) ? FIRST : ap->gonum + 1;
251                 makeid (CLKPROC, INDEX, NAMELENGTH, qrname,
252                         (struct nodeid *) qrap->qrclock);
253
254                 qrap->qrcontrol = (struct nodeid *) ap->goid;
255         }
256
257
258
259         /*
260          * If this is the master node, then perform the sequential
261          * calculations.
```



```
262          */
263
264         if (ap->gomflag) {
265
266                 auto int nv = NUMVECTORS;
267                 auto int index;
268
269                 /*
270                  *  The following large data structures are declared static
271                  *  so that they won't take up a lot of go-node stack space.
272                  *  Since they are used exclusively by the sequential
273                  *  routines, they do not need to be automatic.
274                  */
275
276                 static COLMAT q, aq;
277                 static MAT t;
278                 static VEC er, ei, rsd;
279                 static TYPEVEC type;
280                 static IVEC rsdx;
281
282
283
284                 /*
285                  *  Perform the calculations.
286                  */
287
288                 srrit (q, aq, atq, DEGREEOFA, &nv, ITVECTORS, EPSILON, MAXIT,
289                         ORTRANDOM, t, er, ei, type, rsd, rsdx);
290
291                 (void) printf ("\n\nNumber of vectors that converged: %d\n\n",
292                         nv);
293                 for (index = 1; index <= NUMVECTORS; index++)
294                         (void) printf ("\t(%15.6e, %15.6e)\n",
295                                 ER (index), EI (index));
296
297                 flush ();
298
299         }
300
301
302         LEAVE ("go", DBOFF);
303         finis ();
304 }
```

```
1   /*
2    *  Markov Chain Example
3    *
4    *  AUTHOR:     John R. Meyer
5    *
6    *  DATE:       6/18/91 23:21:42
7    *
8    *  VERSION:    @(#)markov.c 4.1
9    *
10   *  DESCRIPTION:
11   *
12   *     This file contains the code for the Markov chain example used by this
13   *     program.
14   *
15   *     The contents of this file form part of an appendix to the thesis,
16   *     "A Parallel Implementation of a Simultaneous Iteration Algorithm
17   *     for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
18   *     Matrices", by John R. Meyer and submitted to the faculty of the
19   *     Graduate School of the University of Maryland in partial fulfillment
20   *     of the requirements for the degree of master of science.
21   */
22
23
24  /*
25   *  Binary SCCS identification.
26   */
27
28  #ifndef lint
29  static char sccsid [] = "markov.c 4.1";
30  #endif
31
32
33  /*
34   *  Header file inclusions.
35   */
36
37  #include "setup.h"
38  #include "matrix.h"
39  #include "aq.h"
40  #include "debug.h"
41
42
43  /*
44   *  Local function prototypes.
45   */
46
47  static BOOLEAN ongrid ( /* int, int */ );
48  static float prob ( /* DIRECTION, int, int */ );
49  static int qplace ( /* int, int */ );
50  static void findcoord ( /* int, int *, int * */ );
51  static void makeplace ( /* int, int, struct placeinfo * */ );
52
53
54
55  #define SUM(i,j)        BMATX(sum->blkmat,i,j)
56  #define Q(i,j)          BMATX(q->blkmat,i,j)
57
58
59  void aqmultiply (ap, rowdim, q, sum, lower, upper)
60  /*
61   *  Procedure aqmultiply() performs a multiplication of A with the block
62   *  of Q.  For each row in the block, the following procedure is performed:
63   *  A check is made to see if the new block contains information needed for
64   *  the accumulation of our own sum (this information was stored in the places
65   *  array by aqinit).  If it does, that data is multiplied by the
66   *  probability of jumping to another node and this product is added to the
67   *  accumulated sum.
68   *
69   *  The parameters represent:
70   *
71   *      ap              A pointer to the auxiliary storage of the AQ node.
72   *      rowdim          The row dimension of the block of Q.
73   *      q               A block of the matrix Q that has just arrived at
74   *                      this node to be "multiplied" by A.
75   *      sum             Our own accumulation of a block of Q.
76   *      lower           The lower (leftmost) index of the column of Q to
77   *                      be multiplied.
78   *      upper           The upper (rightmost) index of the column of Q to
79   *                      be multiplied.
80   */
81
82  int rowdim, lower, upper;
83  struct aqaux *ap;
84  BLOCK *q, *sum;
85
86  {
87          auto struct placeinfo *places = ap->aqplaceinfo;
```



```
88          auto int row, qcol;
89          auto int dir;
90
91
92          ENTER ("aqmultiply", DBOFF);
93
94
95          for (row = 0; row < rowdim; row++)
96
97                  /*
98                   * Check if q contains information needed by this node for
99                   * each of the four possible directions.
100                  */
101
102                 for (dir = (int) NORTH; dir <= (int) WEST; dir++) {
103
104                         auto float p = prob ((DIRECTION) dir,
105                                 places[row].vert, places[row].horiz);
106                         auto int index = places[row].neighbors[dir] -
107                                 q->begrow + 1;
108
109                         if (1 <= index && index <= q->nbrow)
110                                 for (qcol = lower; qcol <= upper; qcol++)
111                                         SUM (row + 1, qcol) +=
112                                                 p * Q (row + 1, qcol);
113                 }
114
115
116         LEAVE ("aqmultiply", DBOFF);
117 }
118
119
120 #undef  SUM
121 #undef  Q
122
123
124
125 static BOOLEAN ongrid (v, h)
126 /*
127  *  Function ongrid returns TRUE if the point specified by the given
128  *  coordinates is on the grid, FALSE otherwise.
129  *
130  *  The parameters represent:
131  *
132  *      v,h     The vertical and horizontal coordinates of the point.
133  */
134
135 register int v, h;
136
137 {
138         if (v < 1 || h < 1)
139                 return FALSE;
140         else if (v + h <= GRIDSIZE + 1)
141                 return TRUE;
142         else
143                 return FALSE;
144 }
145
146
147
148 static float prob (direction, v, h)
149 /*
150  *  Procedure prob calculates the probability of jumping from a (v,h)
151  *  position on the grid in a certain direction.
152  *
153  *  The parameters represent:
154  *
155  *      direction       The direction to jump in.
156  *      v               The vertical coordinate of the point
157  *                      (beginning with 1).
158  *      h               The horizontal coordinate of the point
159  *                      (beginning with 1).
160  */
161
162 int v, h;
163 DIRECTION direction;
164
165 {
166         register float p = (float) (v + h - 2) / (float) (GRIDSIZE - 1);
167
168
169         switch (direction) {
170         case NORTH:
171                 p = 1.0 - p;
172                 if (ongrid (v, h + 1))
173                         p /= 2.0;
174                 break;
```

```
175         case SOUTH:
176                 if (ongrid (v, h - 1))
177                         p /= 2.0;
178                 break;
179         case EAST:
180                 p = 1.0 - p;
181                 if (ongrid (v + 1, h))
182                         p /= 2.0;
183                 break;
184         case WEST:
185                 if (ongrid (v - 1, h))
186                         p /= 2.0;
187                 break;
188         default:
189                 srrerror ("prob:  case mismatch.\n");
190         }
191 
192         return p;
193 }
194 
195 
196 
197 void aqinit (aqap)
198 /*
199  *  Procedure aqinit fills an array of place information for rows in
200  *  a certain range.  This information is used by the AQ nodes to determine
201  *  when blocks of Q containing relevant information will pass by.
202  *
203  *  The parameter represents:
204  *
205  *      aqap    A pointer to the auxiliary storage of this AQ node.
206  */
207 
208 struct aqaux *aqap;
209 
210 {
211         auto int rowindex = aqap->aqrowindex;
212         auto int rowdim = aqap->aqrowdim;
213         auto struct placeinfo *places = aqap->aqplaceinfo;
214         auto int index;
215         auto int v, h;
216 
217         for (index = rowindex; index < rowindex + rowdim; index++) {
218                 findcoord (index, &v, &h);
219                 makeplace (v, h, places + index - rowindex);
220         }
221 }
222 
223 
224 
225 static void findcoord (index, v, h)
226 /*
227  *  Procedure findcoord determines the vertical and horizontal coordinates
228  *  of the given index into the A matrix.  In this respect, it is a functional
229  *  opposite of qplace.
230  *
231  *  The parameters represent:
232  *
233  *      index           The given matrix index.
234  *      v               The vertical coordinate of the grid element.
235  *      h               The horizontal coordinate of the grid element.
236  */
237 
238 register int index;
239 int *v, *h;
240 
241 {
242         register int position = 1;
243         register int vert, horiz;
244 
245         for (horiz = 1; horiz <= GRIDSIZE; horiz++)
246                 for (vert = 1; vert + horiz <= GRIDSIZE + 1; vert++)
247                         if (position++ == index) {
248                                 *v = vert;
249                                 *h = horiz;
250                                 return;
251                         }
252 }
253 
254 
255 
256 static void makeplace (v, h, place)
257 /*
258  *  Procedure makeplace stores the relevant place information in the
259  *  place structure for a grid coordinate.
260  *
261  *  The parameters represent:
```



```
262  *
263  *      v               The vertical coordinate of the grid position.
264  *      h               The horizontal coordinate of the grid position.
265  *      place           The place structure to put the information.
266  */
267 
268 register int v, h;
269 register struct placeinfo *place;
270 
271 {
272         place->vert = v;
273         place->horiz = h;
274         place->neighbors[(int) NORTH] = (v + h == GRIDSIZE + 1) ?
275                 0 : qplace (v + 1, h);
276         place->neighbors[(int) SOUTH] = (v == 1) ?
277                 0 : qplace (v - 1, h);
278         place->neighbors[(int) EAST] = (v + h == GRIDSIZE + 1) ?
279                 0 : qplace (v, h + 1);
280         place->neighbors[(int) WEST] = (h == 1) ?
281                 0 : qplace (v, h - 1);
282 }
283 
284 
285 
286 
287 static int qplace (v, h)
288 /*
289  *  Procedure qplace returns an integer index into the A matrix
290  *  corresponding to the position of the grid element placed at (v,h).
291  *
292  *  The parameters represent:
293  *
294  *      v       The vertical index of the element.
295  *      h       The horizontal index of the element.
296  */
297 
298 register int v, h;
299 
300 {
301         register int position = 0;
302         register int column;
303 
304         for (column = 1; column < h; column++)
305                 position += GRIDSIZE + 1 - column;
306 
307         return position + v;
308 }
```

```c
/*
 *  Matrix and Block Arithmetic Routines
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:21:59
 *
 *  VERSION:    @(#)matrix.c 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains code for the basic matrix operations performed by
 *      routines in this package, like addition, multiplication, inner
 *      product, and Cholesky factorization.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */



/*
 *  Binary SCCS identification.
 */

#ifndef lint
static char sccsid [] = "matrix.c 4.1";
#endif



/*
 *  Header file inclusions.
 */

#include <math.h>
#include "setup.h"
#include "debug.h"
#include "matrix.h"
#include "time.h"



#define B(i,j)          BLKREF(*b,i,j)


void initblk (b, begrow, nbrow)
/*
 *  Procedure initblk initialzes the given block, setting the indexing
 *  and length parameters as well as filling the matrix elements with zeros.
 *
 *  The parameters represent:
 *
 *      b               A pointer to the block to be initialized.
 *      begrow          The beginning row index of the block with respect
 *                      to the full matrix.
 *      nbrow           The number of rows in this block.
 */

register BLOCK *b;
int begrow, nbrow;

{
        register int row, column;


        ENTER ("initblk", DBOFF);


        b->begrow = begrow;
        b->nbrow = nbrow;

        for (row = 1; row <= nbrow; row++)
                for (column = 1; column <= COLUMNSOFQ; column++)
                        B (row, column) = 0.0;


        LEAVE ("initblk", DBOFF);
}


#undef  B

```



```c


#define Q(i,j)          BLKREF(*q,i,j)
#define AQ(i,j)         BLKREF(*aq,i,j)


void sumblk (q, aq)
/*
 *  Procedure sumblk() calculates the sum of two blocks of a matrix.
 *
 *  The parameters represent:
 *
 *      q       A pointer to the block of Q.
 *      aq      A pointer to the block of AQ.
 */

register BLOCK *q, *aq;

{
        register int row, column;

        ENTER ("sumblk", DBOFF);


        for (row = 1; row <= aq->nbrow; row++)
                for (column = 1; column <= COLUMNSOFQ; column++)
                        AQ (row, column) += Q (row, column);


        LEAVE ("sumblk", DBOFF);
}


#undef  Q
#undef  AQ



#define Q(i,j)          BLKREF(*q,i,j)
#define AQ(i,j)         BLKREF(*aq,i,j)


void multblk (q, aq)
/*
 *  Procedure multblk() calculates the product of trans(q)*aq in a block-
 *  by-block fashion, leaving the result in aq.
 *
 *  The parameters represent:
 *
 *     q                A pointer to the block of Q.
 *     aq               A pointer to the block of AQ.
 */

BLOCK *q, *aq;

{
        auto float work [COLUMNSOFQ];
        auto int row, column, index;
        auto double sum;


        ENTER ("multblk", DBOFF);


        aq->begrow = 1;
        aq->nbrow = COLUMNSOFQ;

        for (row = 1; row <= COLUMNSOFQ; row++) {
                for (column = 1; column <= COLUMNSOFQ; column++) {
                        sum = (double) 0.0;
                        for (index = 1; index <= q->nbrow; index++)
                                sum += ((double) Q (index, column)) *
                                       ((double) AQ (index, row));
                        work[column - 1] = (float) sum;
                }
                for (column = 1; column <= COLUMNSOFQ; column++)
                        AQ (column, row) = work[column - 1];
        }


        LEAVE ("multblk", DBOFF);
}


#undef  Q
#undef  AQ

```

```
175
176
177 #define Q(i,j)          BLKREF(*q,i,j)
178 #define PRODUCT(i)      SBMATX(product->blkmat,i)
179
180
181 void qttq (q, product)
182 /*
183  *  Procedure qttq calculates the product of the transpose of a block
184  *  of a matrix with the block of the matrix, returning a block of a
185  *  symmetric matrix.
186  *
187  *  The parameters represent:
188  *
189  *      q               A pointer to the block that is to be premultiplied by
190  *                      its own transpose.
191  *      product         A pointer to the result.
192  */
193
194 register BLOCK *q;
195 SYMBLOCK *product;
196
197 {
198         register int row, column;
199         register double sum;
200         auto int index, position;
201
202
203         ENTER ("qttq", DBOFF);
204
205
206         product->begrow = 1;
207         product->nbrow = COLUMNSOFQ;
208
209         PRODUCT (BLOCKDIM * (BLOCKDIM + 1) / 2 - 1) = 0.0;
210         position = 1;
211
212         for (column = 1; column <= COLUMNSOFQ; column++)
213                 for (row = 1; row <= column; row++) {
214                         sum = (double) 0.0;
215                         for (index = 1; index <= q->nbrow; index++)
216                                 sum += ((double) Q (index, row)) *
217                                        ((double) Q (index, column));
218                         PRODUCT (position) = (float) sum;
219                         position++;
220                 }
221
222
223         LEAVE ("qttq", DBOFF);
224 }
225
226
227 #undef  Q
228 #undef  PRODUCT
229
230
231
232 #define Q(i,j)          MATX(q,i,j)
233 #define B(i,j)          BLKREF(*b,i,j)
234
235
236 void mat2blk (q, b, begrow, nbrow)
237 /*
238  *  Procedure mat2blk places a section of a full matrix into a block,
239  *  initializing all indexing parameters.
240  *
241  *  The parameters represent:
242  *
243  *      q               A pointer to the full matrix.
244  *      b               A pointer to the block to be filled.
245  *      begrow          The beginning row of the block relative to
246  *                      the full matrix.
247  *      nbrow           The number of rows in the block.
248  */
249
250 register COLMAT q;
251 register BLOCK *b;
252 int begrow, nbrow;
253
254 {
255         register int row, column;
256
257
258         ENTER ("mat2blk", DBOFF);
259
260
261         b->begrow = begrow;
```



```
262         b->nbrow = nbrow;
263
264         for (row = 1; row <= nbrow; row++)
265                 for (column = 1; column <= COLUMNSOFQ; column++)
266                         B (row, column) = Q (row + begrow - 1, column);
267
268
269         LEAVE ("mat2blk", DBOFF);
270 }
271
272 #undef  Q
273 #undef  B
274
275
276
277 #define Q(i,j)          MATX(q,i,j)
278 #define B(i,j)          BLKREF(*b,i,j)
279
280
281 void blk2mat (b, q)
282 /*
283  *  Procedure blk2mat places the information in a block back into the
284  *  appropriate place in a full matrix.  This is more or less the reverse
285  *  operation of mat2blk.
286  *
287  *  The parameters represent:
288  *
289  *      b               A pointer to the block to be transferred.
290  *      q               A pointer to the matrix to be filled.
291  */
292
293 register BLOCK *b;
294 register COLMAT q;
295
296 {
297         register int row, column;
298
299
300         ENTER ("blk2mat", DBOFF);
301
302
303         for (row = 1; row <= b->nbrow; row++)
304                 for (column = 1; column <= COLUMNSOFQ; column++)
305                         Q (row + b->begrow - 1, column) = B (row, column);
306
307
308         LEAVE ("blk2mat", DBOFF);
309 }
310
311
312 #undef  Q
313 #undef  B
314
315
316
317
318 #define SUM(i)          SBLKREF(*sum,i)
319 #define B(i)            SBLKREF(*b,i)
320
321
322 void sblksum (sum, b)
323 /*
324  *  Procedure sblksum calculates the sum of two symmetric blocks.
325  *
326  *  The parameters represent:
327  *
328  *      sum             The symmetric block that is the left addend
329  *                      and will receive the sum.
330  *      b               The other symmetric block addend.
331  */
332
333 register SYMBLOCK *sum, *b;
334
335 {
336         register int index;
337
338         for (index = 1; index <= sizeof (SBMAT) / sizeof (float); index++)
339                 SUM (index) += B (index);
340 }
341
342
343 #undef  SUM
344 #undef  B
345
346
347
348
```

```
349 #define Q(i,j)          BLKREF(*q,i,j)
350
351
352 void blkprt (q, message)
353 /*
354  *  Procedure blkprt prints out the contents of the given block of
355  *  a matrix.
356  *
357  *  The parameters represent:
358  *
359  *      q               A pointer to the block that is to be printed.
360  *      message         An optional message to be printed.
361  */
362
363 BLOCK *q;
364 char *message;
365
366 {
367         auto int row, column;
368
369
370         ENTER ("blkprt", DBOFF);
371
372
373         if (message != NULL) {
374                 (void) printf ("%s", message);
375                 flush ();
376         }
377
378         (void) printf ("begrow: %d  nbrow: %d\n\n", q->begrow, q->nbrow);
379         flush ();
380
381         for (row = 1; row <= q->nbrow; row++) {
382                 for (column = 1; column <= COLUMNSOFQ; column++)
383                         (void) printf ("%15.6e\t", Q (row, column));
384                 (void) printf ("\n");
385                 flush ();
386         }
387
388
389         LEAVE ("blkprt", DBOFF);
390 }
391
392
393 #undef  Q
394
395
396
397 float dot (length, vec1, vec2)
398 /*
399  *  Function dot calculates the dot product (inner product) of two vectors.
400  *
401  *  The parameters represent:
402  *
403  *      length          The length of the two vectors.
404  *      vec1, vec2      The two vectors.
405  */
406
407 int length;
408 float vec1 [], vec2 [];
409
410 {
411         register double sum = (double) 0.0;
412         register int index;
413
414         for (index = 0; index < length; index++)
415                 sum += ((double) vec1[index]) * ((double) vec2[index]);
416
417         return (float) sum;
418 }
419
420
421
422
423 #define AP(i)           SBMATX(ap,i)
424
425
426 void cholesky (ap, degree, info)
427 /*
428  *  Procedure cholesky calculates the Cholesky decomposition of a matrix.
429  *  Procedure cholesky is a C language translation of the FORTRAN routine SPPFA
430  *  from the LINPACK linear algebra package.
431  *
432  *  The parameters represent:
433  *
434  *      ap              The packed form of a symmetric matrix A.  The columns
435  *                      of the upper triangle are stored sequentially in a one-
```



```
436  *                      dimensional array.  On return, ap contains an upper
437  *                      triangular matrix R such that A = TRANS(R) * R.
438  *      degree          The order of the matrix.
439  *      info            A return code that is zero for a normal return or an
440  *                      integer K such that the leading minor of order K is not
441  *                      positive definite.
442  */
443
444 SBMAT ap;
445 int degree, *info;
446
447 {
448         auto int position = 0;
449         auto int column;
450         auto double sum;
451
452
453
454         ENTER ("cholesky", DBOFF);
455
456
457         for (column = 1; column <= degree; column++) {
458
459                 *info = column;
460                 sum = (double) 0.0;
461
462                 {
463                         auto int index = position;
464                         auto int inner = 0;
465                         auto int row;
466
467
468                         for (row = 1; row < column; row++) {
469
470                                 auto float temp;
471
472                                 index++;
473                                 temp = AP (index) - dot (row - 1,
474                                         &AP (inner + 1), &AP (position + 1));
475                                 inner += row;
476                                 temp /= AP (inner);
477                                 AP (index) = temp;
478                                 sum += (double) (temp * temp);
479                         }
480                 }
481
482                 position += column;
483                 sum = ((double) AP (position)) - sum;
484                 if (sum <= (double) 0.0) {
485                         LEAVE ("cholesky", DBOFF);
486                         return;
487                 }
488                 AP (position) = sqrt ((float) sum);
489         }
490
491         *info = 0;
492
493
494         LEAVE ("cholesky", DBOFF);
495 }
496
497
498 #undef  AP
499
500
501
502
503 #define NEWQ(i,j)       BLKREF(*newq,i,j)
504 #define R(i)            SBLKREF(*r,i)
505 #define Q(i,j)          BLKREF(*q,i,j)
506
507
508 void solve (newq, r, q)
509 /*
510  *  Procedure solve solves the system NEWQ * R = Q for NEWQ, where NEWQ
511  *  are blocks of a full matrix and R is a block of a symmetric matrix.
512  *
513  *  The parameters represent:
514  *
515  *      newq            The solution of NEWQ * R = Q
516  *      r               A block of a symmetric matrix.
517  *      q               The old value of Q.
518  */
519
520 BLOCK *newq, *q;
521 SYMBLOCK *r;
522
```

```
523 {
524         auto int position = 1;
525         auto int row, column;
526 
527         newq->begrow = q->begrow;
528         newq->nbrow = q->nbrow;
529 
530         for (column = 1; column <= COLUMNSOFQ; column++) {
531                 for (row = 1; row <= q->nbrow; row++)
532                         NEWQ (row, column) = (Q (row, column) - dot (column - 1,
533                                 &NEWQ (row, 1), &R (position))) /
534                                         R (position + column - 1);
535                 position += column;
536         }
537 }
538 
539 
540 #undef  NEWQ
541 #undef  R
542 #undef  Q
```

```
/*
 *  QR Node Program
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:22:03
 *
 *  VERSION:    @(#)qr.c 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains the code for the node program which calculates
 *      the QR reorthogonalization.
 *
 *      The major steps in this calculation are:
 *
 *              - Pass the neighboring blocks of Q around the ring.
 *
 *              - Collect this node's block of Q.
 *                                     T
 *              - Calculate the product Q Q for this node's block.
 *
 *              - If binary summation is being performed:
 *
 *                      - Receive the appropriate number of partial
 *                        sums from the clockwise neighbor and add it
 *                        to this node's product.
 *
 *                      - If this node is the master node, compute the
 *                        Cholesky factorization (R) of the total sum.
 *
 *                      - Pass R around the ring of processors.
 *
 *              - Otherwise,
 *
 *                      - Circulate the cross-product around the ring
 *                        of processors, summing as we go along.
 *
 *                      - Compute the Cholesky factorization (R) of the
 *                        cross-product.
 *
 *              - Solve our own system of equations for the new value
 *                of Q and pass the blocks around the ring.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */



/*
 *  Binary SCCS identification.
 */

#ifndef lint
static char sccsid [] = "qr.c 4.1";
#endif



/*
 *  Header file inclusions.
 */

#include "domparam.h"
#include "domstruct.h"
#include "domdec.h"

#include "setup.h"
#include "debug.h"
#include "matrix.h"
#include "qr.h"
#include "time.h"




/* ARGSUSED */


void qr (nd, sysp)
/*
 *  Procedure qr is the node program for the QR reorthogonalization
```



```
 *  node.
 *
 *  The parameters represent:
 *
 *      nd              A pointer to the QR node structure
 *      sysp            A pointer to the DOMINO system variables
 */

struct node *nd;
int *sysp;

{
        auto struct qraux *ap = (struct qraux *) nd->auxp;
                                        /* Auxiliary storage pointer */
        auto BLOCK q;                   /* Block of Q matrix */
        auto BLOCK *bq;                 /* Pointer to block array */
        auto BLOCK newq;                /* Block of new Q matrix */
        auto BLOCK temp;                /* Temporary block storage */
        auto SYMBLOCK xmult;            /* Cross product */
        auto SYMBLOCK r;                /* The R matrix */
        auto SYMBLOCK stemp;            /* Temporary block storage */
        auto int npass;                 /* Number of times to pass blocks */
        auto int nadd;                  /* Number of times to add blocks */
        auto int chinfo;                /* Cholesky success indicator */
        auto long int sclk, tclk;       /* Clockwise transmission time */
        auto long int sqtq, tqtq;       /* QTTQ calculation time */
        auto long int srot, trot;       /* Rotation time */
        auto long int sclc, tclc;       /* QR calculation time */
        auto long int schl, tchl;       /* Cholesky calculation time */
        auto long int sslv, tslv;       /* Solve calculation time */
        auto long int scnt, tcnt;       /* Counter-clockwise time */
        auto long int stot, ttot;       /* Total time */


        ENTER ("qr", DBOFF);


        /*
         * Get the pointer to the block array if we're the master node.
         */

        if (ap->qrmflag) {
                request (ap->qrcontrol, (int *) &bq);
                pause ();
        }
        TIMERSTART (FALSE, "starting total timer", stot);


        /*
         * Pass the neighboring blocks of Q around the ring and
         * collect our block.
         */

        TIMERSTART (FALSE, "starting QR clockwise timer", sclk);
        for (npass = LAST - ap->qrnum; npass > 0; npass--)
                if (ap->qrmflag)
                        sendn ((struct nodeid *) ap->qrclock, BLOCKSIZE,
                                (int *) (bq + npass));
                else {
                        request ((struct nodeid *) ap->qrcounter,
                                (int *) &temp);
                        pause ();
                        sendn ((struct nodeid *) ap->qrclock, BLOCKSIZE,
                                (int *) &temp);
                }
        TIMEREND (FALSE, "ending QR clockwise timer", sclk, tclk);

        if (ap->qrmflag)
                q = bq[0];
        else {
                request ((struct nodeid *) ap->qrcounter, (int *) &q);
                pause ();
        }


        /*
         *  Calculate the product  TRANS(Q) * Q.
         */

        TIMERSTART (FALSE, "starting QR QTTQ timer", sqtq);
        qttq (&q, &xmult);
        TIMEREND (FALSE, "ending QR QTTQ timer", sqtq, tqtq);



#if BINSUM
```

```
175
176         /*
177          * Receive the appropriate number of partial sums from the
178          * right and accumulate.
179          */
180
181         tclc = 0;
182         TIMERSTART (FALSE, "starting binary send timer", srot);
183         for (nadd = ap->qrnadd; nadd > 0; nadd--) {
184                 request ((struct nodeid *) ap->qrclock, (int *) &stemp);
185                 pause ();
186                 TIMERSTART (FALSE, "starting binary calc timer", sclc);
187                 sblksum (&xmult, &stemp);
188                 TIMERPAUSE (FALSE, "stopping binary calc timer", sclc, tclc);
189         }
190         TIMEREND (FALSE, "ending binary send timer", srot, trot);
191
192
193         /*
194          * If this is the master node, compute the Cholesky
195          * factorization of the total accumulated sum.
196          */
197
198         if (ap->qrmflag) {
199
200                 TIMERSTART (FALSE, "starting Cholesky timer", schl);
201                 cholesky (xmult.blkmat, xmult.nbrow, &chinfo);
202                 TIMEREND (FALSE, "ending Cholesky timer", schl, tchl);
203
204                 if (chinfo != 0)
205                         srrerror ("qr:  cholesky failed.\n");
206
207         } else {
208
209                 /*
210                  *  There is no need to place timing statements here
211                  *  since it is the master processor that requires
212                  *  the most time.
213                  */
214
215                 sendn ((struct nodeid *) ap->qrcounter, SBSIZE, (int *) &xmult);
216                 for (npass = ap->qrnpass; npass > 0; npass--) {
217                         request ((struct nodeid *) ap->qrclock, (int *) &stemp);
218                         pause ();
219                         sendn ((struct nodeid *) ap->qrcounter, SBSIZE,
220                                 (int *) &stemp);
221                 }
222         }
223
224
225         /*
226          * Pass R along the ring so that each processor gets a copy.
227          */
228
229         if (ap->qrmflag)
230                 sendn ((struct nodeid *) ap->qrclock, SBSIZE, (int *) &xmult);
231         else {
232                 request ((struct nodeid *) ap->qrcounter, (int *) &r);
233                 pause ();
234                 if (!ISLAST (ap->qrnum))
235                         sendn ((struct nodeid *) ap->qrclock, SBSIZE,
236                                 (int *) &r);
237         }
238
239
240         /*
241          * Solve our own system NEWQ * R = Q.
242          */
243
244         TIMERSTART (FALSE, "starting solve timer", sslv);
245         solve (&newq, ap->qrmflag ? &xmult : &r, &q);
246         TIMEREND (FALSE, "ending solve timer", sslv, tslv);
247
248
249
250 #else /* BINSUM */
251
252
253         /*
254          * Circulate the cross-product blocks around the ring,
255          * summing as we go along.
256          */
257
258         tclc = 0;
259         TIMERSTART (FALSE, "starting non-binary send timer", srot);
260         for (npass = LAST; npass > 0; npass--) {
261                 sendn ((struct nodeid *) ap->qrclock, SBSIZE,
```



```
262                         ISLAST (npass)? (int *) &xmult : (int *) &stemp);
263                 request ((struct nodeid *) ap->qrcounter, (int *) &stemp);
264                 pause ();
265                 TIMERSTART (FALSE, "starting non-binary calc timer", sclc);
266                 sblksum (&xmult, &stemp);
267                 TIMERPAUSE (FALSE, "stopping non-binary calc timer", sclc, tclc);
268         }
269         TIMEREND (FALSE, "ending non-binary send timer", srot, trot);
270
271
272         /*
273          * Compute the Cholesky factorization of TRANS(Q) * Q.
274          */
275
276         TIMERSTART (FALSE, "starting Cholesky timer", schl);
277         cholesky (xmult.blkmat, xmult.nbrow, &chinfo);
278         TIMEREND (FALSE, "ending Cholesky timer", schl, tchl);
279
280         if (chinfo != 0)
281                 srrerror ("qr:  cholesky failed.\n");
282
283
284         /*
285          * Solve the system NEWQ * R = Q.
286          */
287
288         TIMERSTART (FALSE, "starting solve timer", sslv);
289         solve (&newq, &xmult, &q);
290         TIMEREND (FALSE, "ending solve timer", sslv, tslv);
291
292
293 #endif /* BINSUM */
294
295
296         /*
297          *  Send our own block of Q around the ring followed by
298          *  the remainder of the blocks.
299          */
300
301         if (ap->qrmflag)
302                 bq[0] = newq;
303         else
304                 sendn ((struct nodeid *) ap->qrcounter, BLOCKSIZE,
305                         (int *) &newq);
306
307         TIMERSTART (FALSE, "starting QR collection timer", scnt);
308         for (npass = 1; npass <= LAST - ap->qrnum; npass++) {
309                 request ((struct nodeid *) ap->qrclock, (int *) &newq);
310                 pause ();
311                 if (ap->qrmflag)
312                         bq[npass] = newq;
313                 else
314                         sendn ((struct nodeid *) ap->qrcounter, BLOCKSIZE,
315                                 (int *) &newq);
316         }
317         TIMEREND (FALSE, "ending QR collection timer", scnt, tcnt);
318
319         TIMEREND (FALSE, "ending total timer", stot, ttot);
320
321
322         /*
323          *  Print out the timing results.
324          */
325
326         TIMERPRINT (TRUE, "Total QR clockwise time", tclk);
327         TIMERPRINT (TRUE, "Total QR QTTQ time", tqtq);
328         TIMERPRINT (TRUE, "Total QR calculation time", tclc);
329         TIMERPRINT (TRUE, "Total QR send time", trot - tclc);
330         TIMERPRINT (TRUE, "Total Cholesky time", tchl);
331         TIMERPRINT (TRUE, "Total solve time", tslv);
332         TIMERPRINT (TRUE, "Total counter-clockwise circulation time", tcnt);
333         TIMERPRINT (TRUE, "Total reorthogonalization time", ttot);
334
335
336         /*
337          *  Notify the go-node that we are finished.
338          */
339
340         if (ap->qrmflag)
341                 sendn ((struct nodeid *) ap->qrcontrol, 0, (int *) NULL);
342
343 }
344
345
346
347
348 #if BINSUM
```

```
349
350
351 void getends (qrnum, left, right)
352 /*
353  *  Procedure getends determines the left and right endpoints of the
354  *  partition containing the given QR number.
355  *
356  *  The parameters represent:
357  *
358  *      qrnum           The QR number whose partition endpoints
359  *                      are desired.
360  *      left            The left endpoint of the partition.
361  *      right           The right endpoint of the partition.
362  */
363
364
365 int qrnum;
366 int *left, *right;
367
368 {
369         register int leftend = 0;
370         register int remaining = NUMVIRPROCS;
371         register int partlength = MAXPROCS;
372         auto BOOLEAN done = FALSE;
373
374
375         while (!done) {
376
377                 /*
378                  *  Find the largest partition size in the remaining interval
379                  *  of processors.
380                  */
381
382                 while (remaining / partlength == 0)
383                         partlength /= 2;
384
385                 /*
386                  *  If the given processor number is within the boundaries of
387                  *  the current interval, then we are done.  Otherwise, adjust
388                  *  the left end of the interval and try again.
389                  */
390
391                 if (leftend <= qrnum && qrnum < leftend + partlength)
392                         done = TRUE;
393                 else {
394                         remaining -= partlength;
395                         leftend += partlength;
396                 }
397         }
398
399         *left = leftend;
400         *right = leftend + partlength - 1;
401 }
402
403
404
405 int numadd (qrnum, left, right)
406 /*
407  *  Function numadd returns the number of additions that a QR node will
408  *  have to perform when it is doing a binary summation.
409  *
410  *  The parameters represent:
411  *
412  *      qrnum           The number of this QR node.
413  *      left            The left endpoint of the current partition.
414  *      right           The right endpoint of the current partition.
415  */
416
417 int qrnum;
418 int left, right;
419
420 {
421         register int number = (qrnum == left && !ISLAST (right)) ? 1 : 0;
422         register int position = qrnum == left ? right + 1 : qrnum - left;
423
424         while (position % 2 == 0 && position != 0) {
425                 position /= 2;
426                 number++;
427         }
428
429         return number;
430 }
431
432
433
434 int numpass (qrnum, left, right)
435 /*
```



```
436  *  Function numpass calculates the number of times the given QR node
437  *  will have to pass partial sums from the right when doing a binary
438  *  summation.
439  *
440  *  The parameters represent:
441  *
442  *      qrnum           The number of the given QR node.
443  *      left            The left endpoint of the partition containing qrnum.
444  *      right           The right endpoint of the partition containing qrnum.
445  */
446
447 register int qrnum;
448 register int left, right;
449
450 {
451         register int npass = (ISLAST (right) || qrnum == left) ? 0 : 1;
452
453
454         while (left != right) {
455
456                 register int mid = (left + right) / 2;
457
458                 /*
459                  *  If the processor is in the left-hand subinterval, increment
460                  *  the number of times it must pass to the left.  Adjust the
461                  *  endpoints of the partition.
462                  */
463
464                 if (left < qrnum && qrnum <= mid) {
465                         npass++;
466                         right = mid;
467                 } else
468                         left = mid + 1;
469         }
470
471         return npass;
472 }
473
474
475 #endif /* BINSUM */
```

```c
/*
 *  Timing Procedures
 *
 *  AUTHOR:     John R. Meyer
 *
 *  DATE:       6/18/91 23:22:14
 *
 *  VERSION:    @(#)send.c 4.1
 *
 *  DESCRIPTION:
 *
 *      This file contains code to time clockwise and counter-clockwise
 *      transmission times for the hardware being used.
 *
 *      The contents of this file form part of an appendix to the thesis,
 *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
 *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
 *      Matrices", by John R. Meyer and submitted to the faculty of the
 *      Graduate School of the University of Maryland in partial fulfillment
 *      of the requirements for the degree of master of science.
 */


#include "domparam.h"
#include "domstruct.h"
#include "domdec.h"
#include "debug.h"
#include "time.h"


/*
 *  Constants for program control.
 */

#define MAXITS          10
#define PHI_ITERATIONS  1000
#define MAXMSGLEN       1000
#define START_LEN       1
#define INCREMENT       10
#define CLOCKWISE       FALSE


/*
 *  Definitions for different versions of DOMINO.
 */

#define NQENTS          nqents
#define NLENTS          nlents
#define STOREX          storex
#define QPOOLP          qpoolp
#define LPOOLP          lpoolp
#define STORE           store
#define SELFADDR        selfaddr
#define NODE            node
#define NNODES          nnodes


/*
 *  DOMINO-related constants.
 */

#define NUMQELMNTS      10000    /* Number of queue elements */
#define NUMLISTELMNTS   10000    /* Number of list elements */
#define FLOATSIZE       ((sizeof (float) / sizeof (int)))

/*
 *  Processor and node identification constants.
 */

#define NEIGHBOR        ((SELFADDR == 1) ? 2 : 1)
#define NAMELENGTH      2

static int neighbor [2 * NAMELENGTH + 3];
static int name [NAMELENGTH] = { 0 };



void boot ();
void go ();


/*
 *  The processor table is an array containing the actual frob numbers
 *  corresponding to the SELFADDR value for the processors on the
 *  machine.  For example, the index of the element containing 1 is
 *  the SELFADDR for frob 1.
 */
```



```c

unsigned short int proctabl [] = {
        0x00ff, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007,
        0x0008, 0x0009, 0x000a, 0x000b, 0x000c, 0x000d, 0x000e, 0x000f
};



/* ARGSUSED */


void boot (nd, sysp)
struct node *nd;                        /* Self node structure */
int *sysp;                              /* System pointer */

{

        /*
         * Initialize the queue pool.
         */

        {
                auto int element;
                auto struct qentry *qstart;

                NQENTS = NUMQELMNTS;
                if (STOREX + NQENTS * QENTSIZE >= STORSIZE)
                        error ("\nboot:  too many queue entries.\n");
                QPOOLP = qstart = (struct qentry *) STORE + STOREX;
                for (element = 1; element <= NQENTS; element++)
                        if (element != NQENTS) {
                                QPOOLP->nextent = QPOOLP + 1;
                                QPOOLP++;
                        } else
                                QPOOLP->nextent = NULL;
                QPOOLP = qstart;
                STOREX += NQENTS * QENTSIZE;
        }


        /*
         * Initialize the list pool.
         */

        {
                auto int element;
                auto struct listentry *lstart;

                NLENTS = NUMLISTELMNTS;
                if (STOREX + NLENTS * LENTSIZE >= STORSIZE)
                        error ("\nboot:  too many list entries.\n");
                LPOOLP = lstart = (struct listentry *) STORE + STOREX;
                for (element = 1; element <= NLENTS; element++)
                        if (element != NLENTS) {
                                LPOOLP->nextent = LPOOLP + 1;
                                LPOOLP++;
                        } else
                                LPOOLP->nextent = NULL;
                LPOOLP = lstart;
                STOREX += NLENTS * LENTSIZE;
        }



        /*
         * Make the go node.
         */

        makenode (USER, NAMELENGTH, name, go, 16, 75000);


        finis ();
}



/* ARGSUSED */


void go (nd, sysp)
struct node *nd;
int *sysp;

{
        static float buffer [MAXMSGLEN] = { 0.0 };
        auto int length;

```

```
175
176         /*
177          *  Make the node identifier for the neighboring node.
178          */
179
180         makeid (NEIGHBOR, NNODES - 1, NAMELENGTH, name, neighbor);
181
182
183         /*
184          *  For each message length, send the message between the two
185          *  processors and tally the time it takes.
186          */
187
188         for (length = START_LEN; length <= MAXMSGLEN; length += INCREMENT) {
189
190                 auto long int total = 0;
191                 auto int loop;
192
193                 /*
194                  *  Print out the status message.  Make sure this is
195                  *  done before the clock starts.
196                  */
197
198                 (void) printf ("Message length %d ...\t", length);
199                 flush();
200
201                 /*
202                  *  Calculate the time it takes to send the message
203                  *  around the belt.
204                  */
205
206                 for (loop = 0; loop < MAXITS; loop++) {
207
208                         auto long int interval = clock ();
209
210 #if CLOCKWISE
211                         if (SELFADDR == 1)
212                                 sendn (neighbor, length * FLOATSIZE, buffer);
213                         else {
214                                 request (neighbor, buffer);
215                                 pause ();
216                         }
217 #else
218                         if (SELFADDR == 1) {
219                                 request (neighbor, buffer);
220                                 pause ();
221                         } else
222                                 sendn (neighbor, length * FLOATSIZE, buffer);
223 #endif /* CLOCKWISE */
224
225                         total += clock () - interval;
226                 }
227
228                 /*
229                  *  Print out the status message.  Make sure this is
230                  *  done after the clock stops.
231                  */
232
233                 (void) printf ("time: %ld ticks\n", total);
234                 flush ();
235         }
236
237
238         finis ();
239 }
```

```
1    /*
2     *  Sequential SRR and Eigenvalue Routines
3     *
4     *  AUTHOR:     John R. Meyer
5     *
6     *  DATE:       6/18/91 23:22:09
7     *
8     *  VERSION:    @(#)srrit.c 4.1
9     *
10    *  DESCRIPTION:
11    *
12    *     This file contains the driver and support routines for the SRR
13    *     iterations.  Each of the major routines in this file are C
14    *     translations of FORTRAN counterparts found in Stewart [10] and
15    *     Smith [6].
16    *
17    *     The contents of this file form part of an appendix to the thesis,
18    *     "A Parallel Implementation of a Simultaneous Iteration Algorithm
19    *     for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
20    *     Matrices", by John R. Meyer and submitted to the faculty of the
21    *     Graduate School of the University of Maryland in partial fulfillment
22    *     of the requirements for the degree of master of science.
23    */
24
25
26
27   /*
28    *  Binary SCCS identification.
29    */
30
31   #ifndef lint
32   static char sccsid [] = "srrit.c 4.1";
33   #endif
34
35
36
37   /*
38    *  Header file inclusions.
39    */
40
41   #include <math.h>
42
43   #include "domparam.h"
44   #include "domstruct.h"
45   #include "domdec.h"
46
47   #include "setup.h"
48   #include "debug.h"
49   #include "matrix.h"
50
51
52
53   #define Q(i,j)          MATX(q,i,j)
54   #define AQ(i,j)         MATX(aq,i,j)
55   #define T(i,j)          MATX(t,i,j)
56   #define ER(i)           VECX(er,i)
57   #define EI(i)           VECX(ei,i)
58   #define RSD(i)          VECX(rsd,i)
59   #define RSDX(i)         IVECX(rsdx,i)
60   #define ORSD(i)         VECX(orsd,i)
61   #define ORSDX(i)        IVECX(orsdx,i)
62   #define TYPE(i)         EVECX(type,i)
63   #define OTYPE(i)        EVECX(otype,i)
64
65   #define INIT            5
66   #define STPFAC          2.0
67   #define ALPHA           1.0
68   #define BETA            1.0
69   #define GRPTOL          0.001
70   #define CNVTOL          0.001
71   #define ORTTOL          2.0
72
73
74
75   /*
76    *  Local function prototypes.
77    */
78
79   static void srrstp ( /* MAT, MAT, void (*)(), int, int, int, MAT, VEC, VEC,
80           TYPEVEC, VEC, VEC, TYPEVEC */ );
81   static void resid ( /* MAT, MAT, MAT, VEC, IVEC, VEC, IVEC, int, int, int,
82           int, int, TYPEVEC */ );
83   void orth ( /* MAT, int, int, int */ );
84   static void group ( /* VEC, VEC, TYPEVEC, VEC, float, int, int, int *, float *,
85           float *, float * */ );
86   static void split ( /* MAT, MAT, int, int, VEC, VEC */ );
87   static void qrstep ( /* MAT, MAT, float *, float *, float *, int, int, int */ );
```



```
88   static void orthes ( /* int, int, int, MAT, VEC */ );
89   static void ortran ( /* int, int, int, MAT, VEC, MAT */ );
90   static void hqr3 ( /* MAT, MAT, int, int, int, float, VEC, VEC, TYPEVEC */ );
91   static void exchng ( /* MAT, MAT, int, int, int, int, float, BOOLEAN * */ );
92   static float cond ( /* MAT, int */ );
93   static float macheps ( /* void */ );
94   static float srrrand ( /* void */ );
95   static float fmax ( /* float, float */ );
96   static double square ( /* double */ );
97   static int min ( /* int, int */ );
98   static int max ( /* int, int */ );
99
100
101
102
103  void srrit (q, aq, atq, n, nv, m, eps, maxit, start, t, er, ei, type, rsd, rsdx)
104  /*
105   *  Author:    G. W. Stewart
106   *  Modified:  John R. Meyer
107   */
108
109  int n, *nv, m, maxit;
110  QINITIAL start;
111  float eps;
112  IVEC rsdx;
113  COLMAT q, aq;
114  MAT t;
115  VEC er, ei, rsd;
116  TYPEVEC type;
117  void (*atq) ();
118
119  {
120          static int dort, dsrr, i, it, j, l, ngrp, nogrp, nxtort, nxtsrr;
121          static float ae, aoe, arsd, aorsd, ctr, octr;
122          static IVEC orsdx;
123          static VEC oer, oei, orsd;
124          static TYPEVEC otype;
125
126
127
128          ENTER ("srrit", DBOFF);
129
130
131          /*
132           * Initialize.
133           */
134
135          l = 1;
136          it = 0;
137
138          for (j = 1; j <= m; j++) {
139                  RSD (j) = 0.0;
140                  RSDX (j) = -1;
141                  TYPE (j) = NO_SUCCESS;
142          }
143
144          switch (start) {
145          case RANDOM:
146          case ORTRANDOM:
147                  for (j = 1; j <= m; j++)
148                          for (i = 1; i <= n; i++)
149                                  Q (i, j) = srrrand ();
150                  if (start == ORTRANDOM)
151                          orth (q, l, m, n);
152                  break;
153          case QCOLUMNS:
154                  break;
155          case ORTQCOLUMNS:
156                  orth (q, l, m, n);
157                  break;
158          default:
159                  srrerror ("srrit:  Q option unaccounted for.\n");
160          }
161
162
163          /*
164           * SRR loop.
165           */
166
167          while (TRUE) {
168
169
170                  srrstp (q, aq, atq, l, m, n, t, er, ei, type, oer, oei, otype);
171                  resid (q, aq, t, rsd, rsdx, orsd, orsdx, l, m, m, n, it, type);
172
173
174                  /*
```

```
175                  * Test for convergence.
176                  */
177 
178                 while (TRUE) {
179 
180                         group (er, ei, type, rsd, GRPTOL, l, m, &ngrp,
181                                 &ctr, &ae, &arsd);
182                         group (oer, oei, otype, orsd, GRPTOL, l, m, &nogrp,
183                                 &octr, &aoe, &aorsd);
184 
185                         if (
186                                 (ngrp != nogrp) ||
187                                 (ngrp == 0) ||
188                                 (fabs (ae - aoe) > ctr * CNVTOL *
189                                         ((float) (RSDX (l) - ORSDX (l)))) ||
190                                 (arsd > eps)
191                                 )
192                                 break;
193 
194                         if ((l += ngrp) > m)
195                                 break;
196                 }
197 
198 
199 
200                 /*
201                  * Exit if the required number of vectors has converged or if
202                  * the iteration count exceeds the maximum number of
203                  * iterations.
204                  */
205 
206                 if (l > *nv)
207                         goto end;
208 
209                 if (it >= maxit) {
210                         (void) printf ("Exceeded maximum iterations (%d)\n",
211                                 maxit); flush();
212                         goto end;
213                 }
214 
215 
216                 /*
217                  * Determine when the next SRR step is to be taken.
218                  */
219 
220                 nxtsrr = min (maxit, max ((int) (STPFAC * it), INIT));
221                 dsrr = nxtsrr - it;
222 
223                 if ((ngrp == nogrp) && (ngrp != 0) && (arsd < aorsd))
224                         dsrr = max (1, (int) (ALPHA + BETA *
225                                 ((float) (ORSDX (l) - RSDX (l))) *
226                                         log (arsd / eps) / log (arsd / aorsd)));
227 
228                 nxtsrr = min (nxtsrr, it + dsrr);
229 
230 
231                 /*
232                  * Determine the interval between orthogonalizations.
233                  */
234 
235                 dort = fmax (1.0, ORTTOL / log10 (cond (t, m)));
236                 nxtort = min (it + dort, nxtsrr);
237 
238                 for (j = 1; j <= m; j++)
239                         for (i = 1; i <= n; i++)
240                                 Q (i, j) = AQ (i, j);
241 
242                 it++;
243 
244 
245                 /*
246                  * Orthogonalization loop.
247                  */
248 
249                 do {
250 
251                         /*
252                          * Power loop.
253                          */
254 
255                         for (; it < nxtort; it++) {
256                                 (*atq) (q, aq, t, l, m, FALSE);
257                                 for (j = l; j <= m; j++)
258                                         for (i = 1; i <= n; i++)
259                                                 Q (i, j) = AQ (i, j);
260                         }
261 
```



```
262                                 orth (q, l, m, n);
263                                 nxtort = min (it + dort, nxtsrr);
264 
265                         } while (it < nxtsrr);
266 
267                 }       /* SRR loop */
268 
269 end:
270                 *nv = l - 1;
271 
272 
273                 LEAVE ("srrit", DBOFF);
274 }
275 
276 
277 #undef Q
278 #undef AQ
279 #undef T
280 #undef ER
281 #undef EI
282 #undef RSD
283 #undef RSDX
284 #undef ORSD
285 #undef ORSDX
286 #undef TYPE
287 #undef OTYPE
288 
289 
290 
291 
292 #define Q(i,j)          MATX(q,i,j)
293 #define AQ(i,j)         MATX(aq,i,j)
294 #define T(i,j)          MATX(t,i,j)
295 #define V(i,j)          MATX(v,i,j)
296 #define ER(i)           VECX(er,i)
297 #define EI(i)           VECX(ei,i)
298 #define OER(i)          VECX(oer,i)
299 #define OEI(i)          VECX(oei,i)
300 #define AP(i)           VECX(ap,i)
301 #define P(i)            VECX(p,i)
302 #define TYPE(i)         EVECX(type,i)
303 #define OTYPE(i)        EVECX(otype,i)
304 
305 
306 
307 static void srrstp (q, aq, atq, l, m, n, t, er, ei, type, oer, oei, otype)
308 /*
309  *  Author:    G. W. Stewart
310  *  Modified:  John R. Meyer
311  */
312 
313 int l, m, n;
314 COLMAT q, aq;
315 MAT t;
316 VEC er, ei, oer, oei;
317 TYPEVEC type, otype;
318 void (*atq) ();
319 
320 {
321         auto float mcheps = macheps ();
322         auto int i, j, k;
323         static MAT v;
324         static VEC p;
325 
326 
327 
328         ENTER ("srrstp", DBOFF);
329 
330 
331         /*
332          * Save the old eigenvalues.
333          */
334 
335         for (j = 1; j <= m; j++) {
336                 OER (j) = ER (j);
337                 OEI (j) = EI (j);
338                 OTYPE (j) = TYPE (j);
339         }
340 
341 
342         /*
343          * Calculate the new T.
344          */
345 
346         (*atq) (q, aq, t, l, m, TRUE);
347 
348 
```

```
        /*
         * Triangularize T.
         */

        orthes (m, l, m, t, p);
        ortran (m, l, m, t, p, v);
        hqr3 (t, v, m, l, m, mcheps, er, ei, type);


        /*
         * Transform Q and AQ.
         */

        for (i = 1; i <= n; i++) {

                static VEC ap;

                for (j = 1; j <= m; j++) {

                        auto float psum = 0.0;
                        auto float apsum = 0.0;

                        for (k = 1; k <= m; k++) {
                                psum += Q (i, k) * V (k, j);
                                apsum += AQ (i, k) * V (k, j);
                        }
                        P (j) = psum;
                        AP (j) = apsum;
                }

                for (j = 1; j <= m; j++) {
                        Q (i, j) = P (j);
                        AQ (i, j) = AP (j);
                }

        }


        LEAVE ("srrstp", DBOFF);
}


#undef Q
#undef AQ
#undef T
#undef V
#undef ER
#undef EI
#undef OER
#undef OEI
#undef AP
#undef P
#undef TYPE
#undef OTYPE


#define Q(i,j)          MATX(q,i,j)
#define AQ(i,j)         MATX(aq,i,j)
#define T(i,j)          MATX(t,i,j)
#define RSD(i)          VECX(rsd,i)
#define ORSD(i)         VECX(orsd,i)
#define RSDX(i)         IVECX(rsdx,i)
#define ORSDX(i)        IVECX(orsdx,i)
#define TYPE(i)         EVECX(type,i)




static void resid (q, aq, t, rsd, rsdx, orsd, orsdx, lower, upper, m, n, it,
        type)
/*
 *  Author:    G. W. Stewart
 *  Modified:  John R. Meyer
 */

int lower, upper, m, n, it;
IVEC rsdx, orsdx;
COLMAT q, aq;
MAT t;
VEC rsd, orsd;
TYPEVEC type;

{
        auto int i, j, k, ku;



```



```
        ENTER ("resid", DBOFF);


        for (j = lower; j <= upper; j++) {
                ORSD (j) = RSD (j);
                ORSDX (j) = RSDX (j);
                RSDX (j) = it;
                ku = (TYPE (j) == REAL_EIG) ? j : min (j + 1, m);
                RSD (j) = 0.0;
                for (i = 1; i <= n; i++) {
                        auto float sum = 0.0;

                        for (k = 1; k <= ku; k++)
                                sum += Q (i, k) * T (k, j);
                        RSD (j) += square (AQ (i, j) - sum);
                }
        }

        for (j = lower; j <= upper; j++) {
                if (TYPE (j) == COMPLEX_POS)
                        RSD (j + 1) = RSD (j) = (RSD (j) + RSD (j + 1)) / 2.0;
                RSD (j) = sqrt (RSD (j));
        }


        LEAVE ("resid", DBOFF);
}


#undef Q
#undef AQ
#undef T
#undef RSD
#undef ORSD
#undef RSDX
#undef ORSDX
#undef TYPE



#define Q(i,j)          MATX(q,i,j)

#define MAXTRY          50
#define TOL             0.5


extern int firstqr [];



/* ARGSUSED */


void orth (q, l, m, n)
int l, m, n;
COLMAT q;

{
        auto int blkidx;
        static BLOCK bq [NUMVIRPROCS];
        auto BLOCK *bqptr = bq;



        ENTER ("orth", DBOFF);


        /*
         *  Split up the matrix into blocks and send the master QR node
         *  the address of the array (mostly for synchronization).
         */

        for (blkidx = 0; blkidx < NUMVIRPROCS; blkidx++)
                mat2blk (q, bq + blkidx, ROWINDEX (blkidx), ROWDIM (blkidx));
        sendn ((struct nodeid *) firstqr, sizeof (BLOCK *), (int *) &bqptr);


        /*
         *  Wait patiently for the calculations to be done and then
         *  glue the Q matrix back together again.
         */

        request ((struct nodeid *) firstqr, (int *) NULL);
        pause ();
        for (blkidx = 0; blkidx < NUMVIRPROCS; blkidx++)
                blk2mat (bq + blkidx, q);

```

```
523
524         LEAVE ("orth", DBOFF);
525 }
526
527
528 #undef Q
529
530
531
532 #define ER(i)           VECX(er,i)
533 #define EI(i)           VECX(ei,i)
534 #define RSD(i)          VECX(rsd,i)
535 #define TYPE(i)         EVECX(type,i)
536
537
538
539
540 static void group (er, ei, type, rsd, grptol, l, m, ngrp, ctr, ae, arsd)
541 /*
542  *  Author:   G. W. Stewart
543  *  Modified: John R. Meyer
544  *
545  *
546  *  Description:
547  *
548  *     Procedure group finds a cluster of complex numbers whose real parts
549  *     are contained in the array "er" and imaginary parts are contained in
550  *     the array "ei".  These numbers are assumed to be stored in descending
551  *     order of magnitude.  "ngrp" is determined as the largest integer
552  *     less than or equal to "m" for which the absolute value E(j) of the
553  *     number ER(j) + EI(j) * i satisfies
554  *
555  *              E(l) - E(l + ngrp - 1) <= GRPTOL / 2.
556  *
557  *     and for which TYPE(l), TYPE(l+1), ..., TYPE(l+ngrp-1) is nonnegative.
558  *     If  "ngrp" = 0, the subroutine returns  "ctr" = "ae" = "arsd" = 0.
559  *     If "ngrp" is nonzero, then "ctr" is set to  (E(l) + E(l+ngrp-1)) / 2,
560  *     "ae" to the average of the numbers RSD(l), RSD(L+1), ..., RSD(l+ngrp-1).
561  */
562
563 int l, m, *ngrp;
564 float grptol, *ctr, *ae, *arsd;
565 VEC er, ei, rsd;
566 TYPEVEC type;
567
568 {
569         auto float mod = sqrt (square (ER (l)) + square (EI (l)));
570
571
572         ENTER ("group", DBOFF);
573
574
575         *ngrp = 0;
576         *ctr = 0.0;
577
578         while (TRUE) {
579
580                 auto int l1 = l + *ngrp;
581                 auto float mod1;
582
583                 if ((l1 > m) || (TYPE (l1) == NO_SUCCESS))
584                         break;
585                 mod1 = sqrt (square (ER (l1)) + square (EI (l1)));
586                 if (fabs (mod - mod1) > grptol * (mod + mod1))
587                         break;
588                 *ctr = (mod + mod1) / 2.0;
589                 *ngrp += (int) (TYPE (l1)) + 1;
590         }
591
592
593         *ae = 0.0;
594         *arsd = 0.0;
595
596         if (*ngrp != 0) {
597
598                 auto int j;
599
600                 for (j = l; j <= l + (*ngrp) - 1; j++) {
601                         *ae += ER (j);
602                         *arsd += square (RSD (j));
603                 }
604                 *ae /= (float) (*ngrp);
605                 *arsd = sqrt (*arsd / ((float) (*ngrp)));
606         }
607         LEAVE ("group", DBOFF);
608 }
609
```



```
610
611 #undef ER
612 #undef EI
613 #undef RSD
614 #undef TYPE
615
616
617
618 #define A(i,j)          MATX(a,i,j)
619 #define V(i,j)          MATX(v,i,j)
620
621
622
623
624 static void split (a, v, n, l, e1, e2)
625 /*
626  *  Author:    G. W. Stewart
627  *  Modified:  John R. Meyer
628  *
629  *
630  *  Description:
631  *
632  *     Given the upper Hessenberg matrix "a" with a 2x2 block starting at
633  *  A(l,l), SPLIT determines if the corresponding eigenvalues are real or
634  *  complex.  If they are real, a rotation is determined that reduces the
635  *  block to upper triangular form with the eigenvalue of largest absolute
636  *  value appearing first.  The rotation is accumulated in "v".  The
637  *  eigenvalues (real or complex) are returned in "e1" and "e2".  The
638  *  parameters in the calling sequence are (starred parameters are altered
639  *  by the procedure):
640  *
641  *     *a       The upper Hessenberg matrix whose 2x2 block is to be split.
642  *     *v       The array in which the splitting transformation is to be
643  *              accumulated.
644  *      n       The order of the matrix "a".
645  *      l       The position of the 2x2 block.
646  *     *e1,*e2  On return, if the eigenvalues are complex, "e1" contains
647  *              their common real part and "e2" contains the positive
648  *              imaginary part.  If the eigenvalues are real, "e1" contains
649  *              the one largest in absolute value and "e2" contains the
650  *              other one.
651  */
652
653 MAT a, v;
654 int n, l;
655 VEC e1, e2;
656
657 {
658         auto int i, j;
659         static float p, q, r, t, u, w, x, y, z;
660
661
662
663         ENTER ("split", DBOFF);
664
665
666         x = A (l + 1, l + 1);
667         y = A (l, l);
668         w = A (l, l + 1) * A (l + 1, l);
669         p = (y - x) / 2.0;
670         q = square (p) + w;
671
672
673         /*
674          * Complex eigenvalue.
675          */
676
677         if (q < 0.0) {
678                 *e1 = p + x;
679                 *e2 = sqrt (-q);
680                 LEAVE ("split", DBOFF);
681                 return;
682         }
683
684         /*
685          * Two real eigenvalues.  Set up transformation.
686          */
687
688         z = sqrt (q);
689         z = (p < 0.0 ? p - z : p + z);
690         r = (z == 0.0) ? 0.0 : -w / z;
691
692         if (fabs (x + z) >= fabs (x + r))
693                 z = r;
694
695         y -= x + z;
696         x = -z;
```

```
        t = A (l, l + 1);
        u = A (l + 1, l);

        if (fabs (y) + fabs (u) > fabs (t) + fabs (x)) {
                q = u;
                p = y;
        } else {
                q = x;
                p = t;
        }

        r = sqrt (square (p) + square (q));
        if (r <= 0.0) {
                *e1 = A (l, l);
                *e2 = A (l + 1, l + 1);
                A (l + 1, l) = 0.0;
                LEAVE ("split", DBOFF);
                return;
        }
        p /= r;
        q /= r;

        /*
         * Premultiply.
         */

        for (j = l; j <= n; j++) {
                z = A (l, j);
                A (l, j) = p * z + q * A (l + 1, j);
                A (l + 1, j) = p * A (l + 1, j) - q * z;
        }

        /*
         * Postmultiply.
         */

        for (i = 1; i <= l + 1; i++) {
                z = A (i, l);
                A (i, l) = p * z + q * A (i, l + 1);
                A (i, l + 1) = p * A (i, l + 1) - q * z;
        }

        /*
         * Accumulate the transformation in "v".
         */

        for (i = 1; i <= n; i++) {
                z = V (i, l);
                V (i, l) = p * z + q * V (i, l + 1);
                V (i, l + 1) = p * V (i, l + 1) - q * z;
        }
        A (l + 1, l) = 0.0;
        *e1 = A (l, l);
        *e2 = A (l + 1, l + 1);
        LEAVE ("split", DBOFF);
        return;
}


#undef A
#undef V



#define A(i,j)          MATX(a,i,j)
#define V(i,j)          MATX(v,i,j)




static void qrstep (a, v, p, q, r, nl, nu, n)
/*
 *  Author:    G. W. Stewart
 *  Modified:  John R. Meyer
 *
 *
 *  Description:
 *
 *     QRSTEP performs one implicit QR step on the upper Hessenberg matrix
 *  "a".  The shift is determined by the numbers "p", "q", and "r", and the
 *  step is applied in rows and columns "nl" through "nu".  The transformations
 *  are accumulated in "v".  The parameters in the calling sequence are
 *  (starred parameters are altered by the procedure):
 *
 *     *a       The upper Hessenberg matrix on which the QR step is be
 *              performed.
```



```
 *     *v       The array in which the transformations are to be calculated.
 *     *p,*q,*r Parameters that determine the shift.
 *      nl      The lower limit of the step.
 *      nu      The upper limit of the step.
 *      n       The order of the matrix "a".
 */

MAT a, v;
int n, nl, nu;
float *p, *q, *r;


{
        auto int i, j, k;

        auto float s, x, y, z;
        auto BOOLEAN last;



        ENTER ("qrstep", DBOFF);


        for (i = nl + 2; i <= nu; i++)
                A (i, i - 2) = 0.0;

        if (nl + 2 != nu)
                for (i = nl + 3; i <= nu; i++)
                        A (i, i - 3) = 0.0;

        for (k = nl; k <= nu - 1; k++) {

                /*
                 * Determine the transformation.
                 */

                last = (k == nu - 1);
                if (k != nl) {
                        *p = A (k, k - 1);
                        *q = A (k + 1, k - 1);
                        *r = 0.0;
                        if (!last)
                                *r = A (k + 2, k - 1);
                        x = fabs (*p) + fabs (*q) + fabs (*r);
                        if (x == 0.0)
                                continue;

                        *p /= x;
                        *q /= x;
                        *r /= x;
                }
                s = sqrt (square (*p) + square (*q) + square (*r));
                if (*p < 0.0)
                        s = -s;
                if (k == nl) {
                        if (nl != 1)
                                A (k, k - 1) = -A (k, k - 1);
                } else
                        A (k, k - 1) = -s * x;
                *p += s;
                x = (*p) / s;
                y = (*q) / s;
                z = (*r) / s;
                *q /= *p;
                *r /= *p;

                /*
                 * Premultiply.
                 */

                for (j = k; j <= n; j++) {
                        *p = A (k, j) + (*q) * A (k + 1, j);
                        if (!last) {
                                *p += (*r) * A (k + 2, j);
                                A (k + 2, j) -= (*p) * z;
                        }
                        A (k + 1, j) -= (*p) * y;
                        A (k, j) -= (*p) * x;
                }

                /*
                 * Postmultiply.
                 */

                j = (k + 3 < nu) ? k + 3 : nu;
                for (i = 1; i <= j; i++) {
                        *p = x * A (i, k) + y * A (i, k + 1);
```

```
                        if (!last) {
                                *p += z * A (i, k + 2);
                                A (i, k + 2) -= (*p) * (*r);
                        }
                        A (i, k + 1) -= (*p) * (*q);
                        A (i, k) -= *p;
                }

                /*
                 * Accumulate the transformation in "v".
                 */

                for (i = 1; i <= n; i++) {
                        *p = x * V (i, k) + y * V (i, k + 1);
                        if (!last) {
                                *p += z * V (i, k + 2);
                                V (i, k + 2) -= (*p) * (*r);
                        }
                        V (i, k + 1) -= (*p) * (*q);
                        V (i, k) -= *p;
                }

        }


        LEAVE ("qrstep", DBOFF);
}


#undef A
#undef V



#define A(i,j)          MATX(a,i,j)
#define ORT(i)          VECX(ort,i)




static void orthes (n, low, high, a, ort)
/*
 *  Author:    EISPACK
 *  Modified:  John R. Meyer
 *
 *
 *  Description:
 *
 *     ORTHES reduces a real general matrix to upper Hessenberg form
 *  using orthogonal similarity transformations.  This reduced form is
 *  used by other subroutines to find the eigenvalues and/or eigenvectors
 *  of the original matrix.  The parameters in the calling sequence are
 *  (starred parameters are altered by the procedure):
 *
 *     n        An integer input variable set equal to the order of
 *              the matrix A.
 *     low,high Two integer input variables indicating the boundary
 *              indices for the balanced matrix.
 *    *a        A single precision real two-dimensional array with
 *              row dimension NM and column dimension at least N.  On
 *              input, A contains the the matrix of order N to be
 *              reduced to Hessenberg form.  On output, A contains
 *              the upper Hessenberg matrix as well as some information
 *              about the orthogonal transformations used in the
 *              reduction.
 *    *ort      A single precision real one-dimensional array of
 *              dimension at least HIGH containing the remaining
 *              information about the orthogonal transformations.
 *              Only components LOW+1 through HIGH are actually used
 *              by ORTHES.
 */

int n, low, high;
MAT a;
VEC ort;

{
        auto int i, j;
        static int m;
        static float g, h, scale;


        ENTER ("orthes", DBOFF);


        for (m = low + 1; m <= high - 1; m++) {
```



```

                h = 0.0;
                ORT (m) = 0.0;

                /*
                 * Scale the column.
                 */

                for (scale = 0.0, i = 1; i <= high; i++)
                        scale += fabs (A (i, m - 1));

                if (scale != 0.0) {

                        for (i = high; i >= m; i--) {
                                ORT (i) = A (i, m - 1) / scale;
                                h += square (ORT (i));
                        }

                        g = (ORT (m) < 0.0) ? sqrt (h) : -sqrt (h);
                        h -= ORT (m) * g;
                        ORT (m) -= g;

                        /*
                         *              T
                         * Form  (I - (U U)/H) * A.
                         */

                        for (j = m; j <= n; j++) {

                                auto float f = 0.0;

                                for (i = high; i >= m; i--)
                                        f += ORT (i) * A (i, j);
                                f /= h;

                                for (i = m; i <= high; i++)
                                        A (i, j) -= f * ORT (i);

                        }

                        /*
                         *              T                    T
                         * Form  (I - (U U)/H) * A * (I - (U U)/H).
                         */

                        for (i = 1; i <= high; i++) {

                                auto float f = 0.0;

                                for (j = high; j >= m; j--)
                                        f += ORT (j) * A (i, j);

                                for (f = f / h, j = m; j <= high; j++)
                                        A (i, j) -= f * ORT (j);

                        }

                        ORT (m) *= scale;
                        A (m, m - 1) = g * scale;

                }
        }


        LEAVE ("orthes", DBOFF);
}


#undef A
#undef ORT



#define A(i,j)          MATX(a,i,j)
#define Z(i,j)          MATX(z,i,j)
#define ORT(i)          VECX(ort,i)




static void ortran (n, low, high, a, ort, z)
/*
 *  Author:    EISPACK
 *  Modified:  John R. Meyer
 *
 *
 *  Description:
```

```c
 *
 *     ORTRAN accumulates the orthogonal similarity transformations
 *  used in the reduction of a real general matrix to upper Hessenberg
 *  form by ORTHES.  The parameters in the calling sequence are
 *  (starred parameters are altered by the procedure):
 *
 *     n        An integer input variable set equal to the order of
 *              the matrix A.  N must not be greater than NM.
 *     low,high Two integer input variables indicating the boundary
 *              indices for the balanced matrix.  If the matrix is
 *              not balanced, set LOW to 1 and HIGH to N.
 *     a        A single precision real two-dimensional array with
 *              row dimension NM and column dimension at least HIGH.
 *              The strict lower triangle of A contains some information
 *              about the orthogonal transformations used in the
 *              reduction to the Hessenberg form.  The remaining upper
 *              part of the matrix is arbitrary.
 *     *ort     A single precision real one-dimensional array of
 *              dimension at least HIGH.  On input, ORT contains
 *              further information about the orthogonal transformations
 *              used in the reduction by ORTHES.  ORT is used for
 *              temporary storage within ORTRAN and is not restored.
 *     *z       A single precision real output two-dimensional array
 *              with row dimension NM and column dimension at least N.
 *              It contains the transformation matrix produced in the
 *              reduction by ORTHES to the upper Hessenberg form.
 */


int n, low, high;
MAT a, z;
VEC ort;


{
        auto int i, j, m;



        ENTER ("ortran", DBOFF);


        for (i = 1; i <= n; i++)
                for (j = 1; j <= n; j++)
                        Z (i, j) = (i == j) ? 1.0 : 0.0;

        for (m = high - 1; m >= low + 1; m--)
                if (A (m, m - 1) != 0.0) {

                        for (i = m + 1; i <= high; i++)
                                ORT (i) = A (i, m - 1);

                        for (j = m; j <= high; j++) {

                                auto float g = 0.0;

                                for (i = m; i <= high; i++)
                                        g += ORT (i) * Z (i, j);

                                g = (g / ORT (m)) / A (m, m - 1);

                                for (i = m; i <= high; i++)
                                        Z (i, j) += g * ORT (i);

                        }

                }
        LEAVE ("ortran", DBOFF);
}


#undef A
#undef Z
#undef ORT




#define MAXITS          30              /* Maximum number of iterations per
                                         * eigenvalue */

#define A(i,j)          MATX(a,i,j)
#define V(i,j)          MATX(v,i,j)
#define ER(i)           VECX(er,i)
#define EI(i)           VECX(ei,i)
#define TYPE(i)         EVECX(type,i)

```



```c



static void hqr3 (a, v, n, nlow, nup, eps, er, ei, type)
/*
 *  Author:    G. W. Stewart
 *  Modified:  John R. Meyer
 *
 *
 *  Description:
 *
 *    HQR3 reduces the upper Hessenberg matrix "a" to quasi-triangular form
 *  by unitary similarity transformations.  The eigenvalues of "a", which are
 *  contained in the 1x1 and 2x2 diagonal blocks of the reduced matrix, are
 *  ordered in descending order of magnitude along the diagonal.  The trans-
 *  formations are accumulated in the array "v".  HQR3 requires the procedures
 *  EXCHNG, QRSTEP, and SPLIT.  The parameters in the calling sequence are
 *  (starred parameters are altered by the procedure):
 *
 *     *a       An array that initially contains the NxN upper Hessenberg
 *              matrix to be reduced.  On return, "a" contains the reduced,
 *              quasi-triangular matrix.
 *     *v       An array that contains a matrix into which the reducing
 *              transformations are to be multiplied.
 *      n       The order of the matrices "a" and "v".
 *      nlow,   Assumed to be zero, and only rows "nlow" through "nup" and
 *      nup     columns "nlow" through "nup" are transformed, resulting in
 *              the calculation of eigenvalues "nlow" through "nup".
 *      eps     A convergence criterion.
 *     *er      An array that on return contains the real parts of the
 *              eigenvalues.
 *     *ei      An array that on return contains the imaginary parts of the
 *              eigenvalues.
 *     *type    An EIG_TYPE array whose i-th entry is
 *                      REAL_EIG        if the i-th eigenvalue is real;
 *                      COMPLEX_POS     if the i-th eigenvalue is complex with
 *                                      positive imaginary part;
 *                      COMPLEX_NEG     if the i-th eigenvalue is complex with
 *                                      negative imaginary part;
 *                      NO_SUCCESS      if the i-th eigenvalue was not
 *                                      calculated successfully.
 */

MAT a, v;
int n, nlow, nup;
VEC ei, er;
float eps;
TYPEVEC type;

{
        auto int mu, nu, i, l;

        auto int it, nl;
        auto float e1, e2, p, q, r, s, t, w, x, y, z;
        auto BOOLEAN fail;



        ENTER ("hqr3", DBOFF);


        for (i = nlow; i <= nup; i++)
                TYPE (i) = NO_SUCCESS;

        t = 0.0;


        /*
         * Main loop.  Find and order the eigenvalues.
         */

        for (nu = nup; nu >= nlow; nu = l - 1) {

                /*
                 * QR loop.  Find negligible elements and perform QR steps.
                 */

                it = 0;

                while (TRUE) {

                        /*
                         * Search back for negligible elements.
                         */

                        for (l = nu; (l != nlow) &&
                                (fabs (A (l, l - 1)) >= eps *
```

```
1219                                    (fabs (A (l - 1, l - 1)) +
1220                                            fabs (A (l, l)))); l--);
1221
1222                    /*
1223                     * Test to see if an eigenvalue or a 2x2 block has
1224                     * been found.
1225                     */
1226
1227                    x = A (nu, nu);
1228                    if (l == nu)
1229                            goto single;
1230                    y = A (nu - 1, nu - 1);
1231                    w = A (nu, nu - 1) * A (nu - 1, nu);
1232                    if (l == nu - 1)
1233                            break;
1234
1235                    if (it == MAXITS)
1236                            goto finish;
1237
1238                    if (it % 10 == 0) {
1239
1240                            /*
1241                             * Ad-hoc shift.
1242                             */
1243
1244                            t += x;
1245                            for (i = nlow; i <= nu; i++)
1246                                    A (i, i) = A (i, i) - x;
1247                            s = fabs (A (nu, nu - 1)) +
1248                                    fabs (A (nu - 1, nu - 2));
1249                            x = 0.75 * s;
1250                            y = x;
1251                            w = -0.4375 * square (s);
1252                    }
1253                    it++;
1254
1255                    /*
1256                     * Look for two consecutive small sub-diagonal
1257                     * elements.
1258                     */
1259
1260                    nl = nu - 2;
1261                    while (TRUE) {
1262                            z = A (nl, nl);
1263                            r = x - z;
1264                            s = y - z;
1265                            p = (r * s - w) / A (nl + 1, nl) +
1266                                    A (nl, nl + 1);
1267                            q = A (nl + 1, nl + 1) - z - r - s;
1268                            r = A (nl + 2, nl + 1);
1269                            s = fabs (p) + fabs (q) + fabs (r);
1270                            p /= s;
1271                            q /= s;
1272                            r /= s;
1273                            if ((nl == 1) ||
1274                                    (fabs (A (nl, nl - 1)) *
1275                                    (fabs (q) + fabs (r)) <=
1276                                            eps * fabs (p) *
1277                                            (fabs (A (nl - 1, nl - 1)) +
1278                                            fabs (z) +
1279                                            fabs (A (nl + 1, nl + 1)))))
1280                                    break;
1281                            nl--;
1282                    }
1283
1284                    /*
1285                     * Perform a QR step between "nl" and "nu".
1286                     */
1287
1288                    qrstep (a, v, &p, &q, &r, nl, nu, n);
1289
1290            }       /* QR loop */
1291
1292            /*
1293             * 2x2 block found.
1294             */
1295
1296            if (nu != nlow + 1)
1297                    A (nu - 1, nu - 2) = 0.0;
1298            A (nu, nu) += t;
1299            A (nu - 1, nu - 1) += t;
1300            TYPE (nu) = REAL_EIG;
1301            TYPE (nu - 1) = REAL_EIG;
1302            mu = nu;
1303
1304            /*
1305             * Loop to position 2x2 block.
```



```
1306             */
1307
1308            while (TRUE) {
1309
1310                    nl = mu - 1;
1311
1312                    /*
1313                     * Attempt to split the blocks into two real
1314                     * eigenvalues.
1315                     */
1316
1317                    split (a, v, n, nl, &e1, &e2);
1318
1319                    /*
1320                     * If the split was successful, go and order the real
1321                     * eigenvalues.
1322                     */
1323
1324                    if (A (mu, mu - 1) == 0.0)
1325                            goto position;
1326
1327                    /*
1328                     * Test to see if the block is properly positioned,
1329                     * and, if not, exchange it.
1330                     */
1331
1332                    if (mu == nup)
1333                            goto get_next;
1334
1335                    if ((mu != nup - 1) && (A (mu + 2, mu + 1) != 0.0)) {
1336
1337                            /*
1338                             * The next block is 2x2.
1339                             */
1340
1341                            if (A (mu - 1, mu - 1) *
1342                                    A(mu, mu) - A(mu - 1, mu) *
1343                                    A(mu, mu - 1) >=
1344                                    A(mu + 1, mu + 1) * A(mu + 2, mu + 2) -
1345                                    A(mu + 1, mu + 2) * A(mu + 2, mu + 1))
1346                                    goto get_next;
1347
1348                            exchng (a, v, n, nl, 2, 2, eps, &fail);
1349                            if (fail) {
1350                                    TYPE (nl) = NO_SUCCESS;
1351                                    TYPE (nl + 1) = NO_SUCCESS;
1352                                    TYPE (nl + 2) = NO_SUCCESS;
1353                                    TYPE (nl + 3) = NO_SUCCESS;
1354                                    goto finish;
1355                            }
1356                            mu += 2;
1357
1358                    } else {
1359
1360                            /*
1361                             * The next block is 1x1.
1362                             */
1363
1364                            if (A (mu - 1, mu - 1) * A (mu, mu) -
1365                                    A (mu - 1, mu) * A (mu, mu - 1) >=
1366                                    square (A (mu + 1, mu + 1)))
1367                                    goto get_next;
1368
1369                            exchng (a, v, n, nl, 2, 1, eps, &fail);
1370                            if (fail) {
1371                                    TYPE (nl) = NO_SUCCESS;
1372                                    TYPE (nl + 1) = NO_SUCCESS;
1373                                    TYPE (nl + 2) = NO_SUCCESS;
1374                                    goto finish;
1375                            }
1376                            mu++;
1377
1378                    }
1379
1380            }       /* while */
1381
1382            /*
1383             * Single eigenvalue found.
1384             */
1385
1386 single:
1387
1388            nl = 0;
1389            A (nu, nu) += t;
1390            if (nu != nlow)
1391                    A (nu, nu - 1) = 0.0;
1392            TYPE (nu) = REAL_EIG;
```

```
1393                mu = nu;
1394
1395                /*
1396                 * Loop to position one or two eigenvalues.
1397                 */
1398
1399 position:
1400
1401                do {
1402
1403                        while (mu != nup) {
1404
1405                                if ((mu == nup - 1) ||
1406                                        (A (mu + 2, mu + 1) == 0.0)) {
1407
1408                                        if (fabs (A (mu, mu)) >=
1409                                                fabs (A (mu + 1, mu + 1)))
1410                                                break;
1411                                        exchng (a, v, n, mu, 1, 1, eps, &fail);
1412                                        mu++;
1413
1414                                } else {
1415
1416                                        if (square (A (mu, mu)) >=
1417                                                A (mu + 1, mu + 1) *
1418                                                A (mu + 2, mu + 2) -
1419                                                A (mu + 1, mu + 2) *
1420                                                A (mu + 2, mu + 1))
1421                                                goto get_next;
1422                                        exchng (a, v, n, mu, 1, 2, eps, &fail);
1423                                        if (fail) {
1424                                                TYPE (mu) = NO_SUCCESS;
1425                                                TYPE (mu + 1) = NO_SUCCESS;
1426                                                TYPE (mu + 2) = NO_SUCCESS;
1427                                                goto finish;
1428                                        } else
1429                                                mu += 2;
1430                                }
1431
1432                        }       /* while */
1433
1434                        mu = nl;
1435                        nl = 0;
1436
1437                } while (mu != 0);
1438
1439
1440                /*
1441                 * Go back and get the next eigenvalue.
1442                 */
1443
1444 get_next:
1445
1446                ;
1447
1448        }       /* for */
1449
1450
1451        /*
1452         * All the eigenvalues have been found and ordered.  Compute their
1453         * values and types.
1454         */
1455
1456 finish:
1457
1458        if (nu >= nlow)
1459                for (i = 1; i <= nu; i++)
1460                        A (i, i) += t;
1461
1462        nu = nup;
1463
1464        do {
1465
1466                if (TYPE (nu) == NO_SUCCESS)
1467                        nu--;
1468
1469                else {
1470
1471                        if ((nu != nlow) && (A (nu, nu - 1) != 0.0)) {
1472
1473                                /*
1474                                 * 2x2 block.
1475                                 */
1476
1477                                split (a, v, n, nu - 1, &e1, &e2);
1478
1479                                if (A (nu, nu - 1) != 0.0) {
```



```
1480                                        ER (nu) = e1;
1481                                        EI (nu - 1) = e2;
1482                                        ER (nu - 1) = ER (nu);
1483                                        EI (nu) = -EI (nu - 1);
1484                                        TYPE (nu - 1) = COMPLEX_POS;
1485                                        TYPE (nu) = COMPLEX_NEG;
1486                                        nu -= 2;
1487                                        continue;
1488                                }
1489                        }
1490
1491                        /*
1492                         * Single root.
1493                         */
1494
1495                        ER (nu) = A (nu, nu);
1496                        EI (nu) = 0.0;
1497                        nu--;
1498
1499                }
1500
1501        } while (nu >= nlow);
1502
1503
1504        LEAVE ("hqr3", DBOFF);
1505 }
1506
1507
1508 #undef A
1509 #undef V
1510 #undef ER
1511 #undef EI
1512 #undef TYPE
1513
1514
1515
1516 #define MAXITS          30              /* Maximum number of iterations to
1517                                          * perform */
1518
1519 #define A(i,j)          MATX(a,i,j)
1520 #define V(i,j)          MATX(v,i,j)
1521
1522
1523
1524
1525 static void exchng (a, v, n, l, b1, b2, eps, fail)
1526 /*
1527  *  Author:    G. W. Stewart
1528  *  Modified:  John R. Meyer
1529  *
1530  *
1531  *  Description:
1532  *
1533  *     Given the upper Hessenberg matrix "a" with consecutive "b1"x"b1" and
1534  *  "b2"x"b2" diagonal blocks ("b1", "b2" >= 2) starting at A(l,l), EXCHNG
1535  *  produces a unitary similarity transformation that exchanges the blocks
1536  *  along with their eigenvalues.  The transformation is accumulated in "v".
1537  *  EXCHNG requires the procedure QRSTEP.  The parameters in the calling
1538  *  sequence are (starred parameters are altered by the procedure):
1539  *
1540  *     *a       The matrix whose blocks are to be interchanged.
1541  *     *v       The array into which the transformations are to be
1542  *              accumulated.
1543  *      n       The order of the matrix "a".
1544  *      l       The position of the blocks.
1545  *      b1      The size of the first block.
1546  *      b2      The size of the second block.
1547  *      eps     A convergence criterion.
1548  *     *fail    A logical variable which is FALSE on a normal return.  If
1549  *              thirty iterations were performed without convergence, "fail"
1550  *              is set to TRUE and the element
1551  *                      A(l+b2,l+b2-1)
1552  *              cannot be assumed zero.
1553  *      na      The first dimension of the array "a".
1554  *      nv      The first dimension of the array "v".
1555  */
1556
1557
1558 MAT a, v;
1559 int n, l, b1, b2;
1560 float eps;
1561 BOOLEAN *fail;
1562
1563
1564 {
1565        auto int i, j;
1566
```

```c
        static int it, m;
        static float p, q, r, s, w, x, y, z;
        static BOOLEAN doflag;


        ENTER ("exchng", DBOFF);


        *fail = FALSE;

        if (b1 != 2) {

                if (b2 != 2) {

                        /*
                         * Interchange 1x1 and 1x1 blocks.
                         */

                        q = A (l + 1, l + 1) - A (l, l);
                        p = A (l, l + 1);
                        r = (p > q) ? p : q;
                        if (r == 0.0) {
                                LEAVE ("exchng", DBOFF);
                                return;
                        }
                        p /= r;
                        q /= r;
                        r = sqrt (square (p) + square (q));
                        p /= r;
                        q /= r;
                        for (j = l, doflag = TRUE; (j <= n) || doflag;
                                j++, doflag = FALSE) {
                                s = p * A (l, j) + q * A (l + 1, j);
                                A (l + 1, j) = p * A (l + 1, j) - q * A (l, j);
                                A (l, j) = s;
                        }
                        for (i = 1, doflag = TRUE; (i <= l + 1) || doflag;
                                i++, doflag = FALSE) {
                                s = p * A (i, l) + q * A (i, l + 1);
                                A (i, l + 1) = p * A (i, l + 1) - q * A (i, l);
                                A (i, l) = s;
                        }
                        for (i = 1, doflag = TRUE; (i <= n) || doflag;
                                i++, doflag = FALSE) {
                                s = p * V (i, l) + q * V (i, l + 1);
                                V (i, l + 1) = p * V (i, l + 1) - q * V (i, l);
                                V (i, l) = s;
                        }
                        A (l + 1, l) = 0.0;
                        LEAVE ("exchng", DBOFF);
                        return;
                }

                /*
                 * Interchange 1x1 and 2x2 blocks.
                 */

                x = A (l, l);
                p = 1.0;
                q = 1.0;
                r = 1.0;
                qrstep (a, v, &p, &q, &r, l, l + 2, n);
                it = 0;
                do {
                        it++;
                        if (it > MAXITS) {
                                *fail = TRUE;
                                LEAVE ("exchng", DBOFF);
                                return;
                        }
                        p = A (l, l) - x;
                        q = A (l + 1, l);
                        r = 0.0;
                        qrstep (a, v, &p, &q, &r, l, l + 2, n);
                } while (fabs (A (l + 2, l + 1)) > eps *
                        (fabs (A (l + 1, l + 1)) + fabs (A (l + 2, l + 2))));
                A (l + 2, l + 1) = 0.0;
                LEAVE ("exchng", DBOFF);
                return;
        }

        /*
         * Interchange 2x2 and "b2"x"b2" blocks.
         */

        m = l + 2;
```



```c
        if (b2 == 2)
                m++;
        x = A (l + 1, l + 1);
        y = A (l, l);
        w = A (l + 1, l) * A (l, l + 1);
        p = 1.0;
        q = 1.0;
        r = 1.0;
        qrstep (a, v, &p, &q, &r, l, m, n);
        it = 0;
        do {
                it++;
                if (it > MAXITS) {
                        *fail = TRUE;
                        LEAVE ("exchng", DBOFF);
                        return;
                }
                z = A (l, l);
                r = x - z;
                s = y - z;
                p = (r * s - w) / A (l + 1, l) + A (l, l + 1);
                q = A (l + 1, l + 1) - z - r - s;
                r = A (l + 2, l + 1);
                s = fabs (p) + fabs (q) + fabs (r);
                p /= s;
                q /= s;
                r /= s;
                qrstep (a, v, &p, &q, &r, l, m, n);
        } while (fabs (A (m - 1, m - 2)) > eps *
                (fabs (A (m - 1, m - 1)) + fabs (A (m - 2, m - 2))));
        A (m - 1, m - 2) = 0.0;
        LEAVE ("exchng", DBOFF);
        return;

}


#undef A
#undef V


#define T(i,j)          MATX(t,i,j)
#define T1(i,j)         MATX(t1,i,j)
#define PVT(i)          IVECX(pvt,i)
#define MULT(i)         VECX(mult,i)




static float cond (t, m)

/*
 *  Author:   G. W. Stewart
 *  Modified: John R. Meyer
 *
 *
 *  Description:
 *
 *     Function cond returns the condition number with respect to the
 *     row-sum norm of the upper Hessenberg matrix t of order m.
 *
 *  The parameters represent:
 *
 *      t       The upper Hessenberg matrix whose condition is desired.
 *      m       The order of t.
 */

int m;
MAT t;

{
        auto float nt = 0.0;

        static int i, j, k;
        static float nt1;
        static IVEC pvt;
        static MAT t1;
        static VEC mult;



        ENTER ("cond", DBOFF);


        for (i = 1; i <= m; i++) {
                auto float ntr = 0.0;
```

```
1741
1742                for (j = max (i - 1, 1); j <= m; j++) {
1743                        T1 (i, j) = T (i, j);
1744                        ntr += fabs (T (i, j));
1745                }
1746                nt = fmax (nt, ntr);
1747        }
1748
1749        for (i = 1; i <= m - 1; i++) {
1750                PVT (i) = 0;
1751                MULT (i) = 0.0;
1752                if (T1 (i + 1, i) != 0.0) {
1753                        if (fabs (T1 (i + 1, i)) > fabs (T1 (i, i))) {
1754                                PVT (i) = 1;
1755                                for (j = i; j <= m; j++) {
1756                                        auto float temp = T1 (i, j);
1757
1758                                        T1 (i, j) = T1 (i + 1, j);
1759                                        T1 (i + 1, j) = temp;
1760                                }
1761                        }
1762                        MULT (i) = T1 (i + 1, i) / T1 (i, i);
1763                        T1 (i + 1, i) = 0.0;
1764                        for (j = i + 1; j <= m; j++)
1765                                T1 (i + 1, j) -= MULT (i) * T1 (i, j);
1766                }
1767        }
1768
1769        for (j = 1; j <= m; j++) {
1770                if (T1 (j, j) == 0.0) {
1771                        LEAVE ("cond", DBOFF);
1772                        return (1.0e+08);
1773                }
1774                T1 (j, j) = 1.0 / T1 (j, j);
1775                if (j != 1)
1776                        for (i = 1; i <= j - 1; i++) {
1777                                auto float sum = 0.0;
1778
1779                                for (k = i; k <= j - 1; k++)
1780                                        sum += T1 (i, k) * T1 (k, j);
1781                                T1 (i, j) = -sum * T1 (j, j);
1782                        }
1783        }
1784
1785        for (j = m - 1; j >= 1; j--) {
1786                if (MULT (j) != 0.0)
1787                        for (i = 1; i <= j + 1; i++)
1788                                T1 (i, j) -= MULT (j) * T1 (i, j + 1);
1789                if (PVT (j) != 0)
1790                        for (i = 1; i <= j + 1; i++) {
1791                                auto float sum = T1 (i, j);
1792
1793                                T1 (i, j) = T1 (i, j + 1);
1794                                T1 (i, j + 1) = sum;
1795                        }
1796        }
1797
1798        nt1 = 0.0;
1799        for (i = 1; i <= m; i++) {
1800                auto float nt1r = 0.0;
1801
1802                for (j = max (1, i - 1); j <= m; j++)
1803                        nt1r += fabs (T1 (i, j));
1804                nt1 = fmax (nt1, nt1r);
1805        }
1806
1807
1808        LEAVE ("cond", DBOFF);
1809        return (nt * nt1);
1810 }
1811
1812
1813 #undef T
1814 #undef T1
1815 #undef PVT
1816 #undef MULT
1817
1818
1819
1820 static float macheps ()
1821 {
1822        static BOOLEAN done = FALSE;
1823        static float epsilon;
1824
1825
1826        if (!done) {
1827                for (epsilon = 1.0; epsilon + 1.0 > 1.0; epsilon /= 2.0)
```



```
1828                        ;
1829                done = TRUE;
1830        }
1831        return epsilon;
1832 }
1833
1834
1835
1836 static float srrrand ()
1837 /*
1838  *  Author:    G. W. Stewart
1839  *  Modified:  John R. Meyer
1840  */
1841
1842 {
1843        static int seed = 69;
1844
1845        seed = (4621 * seed + 2113) % 10000;
1846        return (float) seed / 10000.0;
1847 }
1848
1849
1850
1851 static float fmax (x, y)
1852 float x, y;
1853
1854 {
1855        return x > y ? x : y;
1856 }
1857
1858
1859
1860 static double square (x)
1861 double x;
1862
1863 {
1864        return x * x;
1865 }
1866
1867
1868
1869 static int min (i, j)
1870 int i, j;
1871
1872 {
1873        return i < j ? i : j;
1874 }
1875
1876
1877
1878 static int max (i, j)
1879 int i, j;
1880
1881 {
1882        return i > j ? i : j;
1883 }
```

```
1   /*
2    *  Timing Procedures
3    *
4    *  AUTHOR:     John R. Meyer
5    *
6    *  DATE:       6/18/91 23:22:17
7    *
8    *  VERSION:    @(#)time.c 4.1
9    *
10   *  DESCRIPTION:
11   *
12   *      This file contains code for the calculations of sigma and tau,
13   *      the values for the startup and transmission times for the
14   *      hardware being used.
15   *
16   *      The contents of this file form part of an appendix to the thesis,
17   *      "A Parallel Implementation of a Simultaneous Iteration Algorithm
18   *      for Calculation of Nested Invariant Subspaces of Large Non-Hermitian
19   *      Matrices", by John R. Meyer and submitted to the faculty of the
20   *      Graduate School of the University of Maryland in partial fulfillment
21   *      of the requirements for the degree of master of science.
22   */
23
24
25  #include "domparam.h"
26  #include "domstruct.h"
27  #include "domdec.h"
28  #include "debug.h"
29  #include "time.h"
30
31
32  /*
33   *  Constants for program control.
34   */
35
36  #define BOTHSEND        FALSE
37  #define MAXITS          10
38  #define MAXMSGLEN       2000
39  #define START_LEN       1
40  #define INCREMENT       10
41
42
43  /*
44   *  Definitions for different versions of DOMINO.
45   */
46
47  #define NQENTS          nqents
48  #define NLENTS          nlents
49  #define STOREX          storex
50  #define QPOOLP          qpoolp
51  #define LPOOLP          lpoolp
52  #define STORE           store
53  #define SELFADDR        selfaddr
54  #define NODE            node
55  #define NNODES          nnodes
56
57
58  /*
59   *  DOMINO-related constants.
60   */
61
62  #define NUMQELMNTS      10000    /* Number of queue elements */
63  #define NUMLISTELMNTS   10000    /* Number of list elements */
64  #define FLOATSIZE       ((sizeof (float) / sizeof (int)))
65
66  /*
67   *  Processor and node identification constants.
68   */
69
70  #define NEIGHBOR        ((SELFADDR == 1) ? 2 : 1)
71  #define NAMELENGTH      2
72
73  static int neighbor [2 * NAMELENGTH + 3];
74  static int name [NAMELENGTH] = { 0 };
75
76
77
78  void boot ();
79  void go ();
80
81
82  /*
83   *  The processor table is an array containing the actual frob numbers
84   *  corresponding to the SELFADDR value for the processors on the
85   *  machine.  For example, the index of the element containing 1 is
86   *  the SELFADDR for frob 1.
87   */
```



```
88
89  unsigned short int proctabl [] = {
90          0x00ff, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007,
91          0x0008, 0x0009, 0x000a, 0x000b, 0x000c, 0x000d, 0x000e, 0x000f
92  };
93
94
95
96  /* ARGSUSED */
97
98
99  void boot (nd, sysp)
100 struct node *nd;                        /* Self node structure */
101 int *sysp;                              /* System pointer */
102
103 {
104
105         /*
106          * Initialize the queue pool.
107          */
108
109         {
110                 auto int element;
111                 auto struct qentry *qstart;
112
113                 NQENTS = NUMQELMNTS;
114                 if (STOREX + NQENTS * QENTSIZE >= STORSIZE)
115                         error ("\nboot:  too many queue entries.\n");
116                 QPOOLP = qstart = (struct qentry *) STORE + STOREX;
117                 for (element = 1; element <= NQENTS; element++)
118                         if (element != NQENTS) {
119                                 QPOOLP->nextent = QPOOLP + 1;
120                                 QPOOLP++;
121                         } else
122                                 QPOOLP->nextent = NULL;
123                 QPOOLP = qstart;
124                 STOREX += NQENTS * QENTSIZE;
125         }
126
127
128         /*
129          * Initialize the list pool.
130          */
131
132         {
133                 auto int element;
134                 auto struct listentry *lstart;
135
136                 NLENTS = NUMLISTELMNTS;
137                 if (STOREX + NLENTS * LENTSIZE >= STORSIZE)
138                         error ("\nboot:  too many list entries.\n");
139                 LPOOLP = lstart = (struct listentry *) STORE + STOREX;
140                 for (element = 1; element <= NLENTS; element++)
141                         if (element != NLENTS) {
142                                 LPOOLP->nextent = LPOOLP + 1;
143                                 LPOOLP++;
144                         } else
145                                 LPOOLP->nextent = NULL;
146                 LPOOLP = lstart;
147                 STOREX += NLENTS * LENTSIZE;
148         }
149
150
151
152         /*
153          * Make the go node.
154          */
155
156         makenode (USER, NAMELENGTH, name, go, 16, 75000);
157
158
159         finis ();
160 }
161
162
163
164 /* ARGSUSED */
165
166
167 void go (nd, sysp)
168 struct node *nd;
169 int *sysp;
170
171 {
172         static float buffer [MAXMSGLEN] = { 0.0 };
173         auto float sumxi = 0.0, sumyi = 0.0, sumxiyi = 0.0, sumxisq = 0.0;
174         auto int numitems = 0;
```

```
175         auto float sigma, tau;
176         auto int length;
177 
178 
179         /*
180          *  Make the node identifier for the neighboring node.
181          */
182 
183         makeid (NEIGHBOR, NNODES - 1, NAMELENGTH, name, neighbor);
184 
185 
186         /*
187          *  For each message length, send the message between the two
188          *  processors and tally the time it takes.
189          */
190 
191         for (length = START_LEN; length <= MAXMSGLEN; length += INCREMENT) {
192 
193                 auto float total = 0.0;
194                 auto int loop;
195 
196                 /*
197                  *  Print out the status message.  Make sure this is
198                  *  done before the clock starts.
199                  */
200 
201                 (void) printf ("Message length %d ...\t", length);
202                 flush();
203 
204                 /*
205                  *  Calculate the time it takes to send the message
206                  *  from processor 1 to processor 2 and then from
207                  *  processor 2 to processor 1.
208                  */
209 
210                 for (loop = 0; loop < MAXITS; loop++) {
211 
212                         auto float interval = (float) clock ();
213 
214 #if BOTHSEND
215                         sendn (neighbor, length * FLOATSIZE, buffer);
216                         request (neighbor, buffer);
217                         pause ();
218 #else
219                         if (SELFADDR == 1) {
220                                 sendn (neighbor, length * FLOATSIZE, buffer);
221                                 request (neighbor, buffer);
222                                 pause ();
223                         } else {
224                                 request (neighbor, buffer);
225                                 pause ();
226                                 sendn (neighbor, length * FLOATSIZE, buffer);
227                         }
228 #endif /* BOTHSEND */
229 
230                         total += clock () - interval;
231                 }
232 
233                 /*
234                  *  Divide the total time it took by (2 * MAXITS) to
235                  *  account for the message having been sent MAXITS
236                  *  times (MAXITS) back and forth (2).
237                  */
238 
239                 total /= 2 * MAXITS;
240                 sumyi += total;
241                 sumxi += length;
242                 sumxiyi += length * total;
243                 sumxisq += length * length;
244                 numitems++;
245 
246                 /*
247                  *  Print out the status message.  Make sure this is
248                  *  done after the clock stops.
249                  */
250 
251                 (void) printf ("time: %7.0f usecs\t", TOMICROSECS(total));
252                 tau = (numitems * sumxiyi - sumxi * sumyi) /
253                         (numitems * sumxisq - sumxi * sumxi);
254                 sigma = (sumxisq * sumyi - sumxiyi * sumxi) /
255                         (numitems * sumxisq - sumxi * sumxi);
256                 (void) printf ("sigma = %7.0f   tau = %7.0f\n",
257                         TOMICROSECS(sigma), TOMICROSECS(tau));
258                 flush ();
259         }
260 
261 
```



```
262         finis ();
263 }
```

APPENDIX 3

Related Methods

This appendix mentions some other methods of finding eigenvalues and eigenvectors of matrices suited to problems involving large matrices in a parallel environment.

One of the earliest methods was the *power iteration*, which finds the dominant absolute eigenvalue and corresponding eigenvector of a general matrix according to the iteration

$$x_{\nu+1} = Ax_\nu, \quad \nu = 0, 1, 2, \cdots$$

It can be shown that this iteration will produce a sequence of vectors such that

$$\lim_{\nu\to\infty} \frac{\|x_{\nu+1}\|_\infty}{\|x_\nu\|_\infty} = |\lambda_1|$$

and that

$$\lim_{\nu\to\infty} \frac{x_\nu}{\|x_\nu\|_\infty}$$

is its associated eigenvector. This method is in essence the simultaneous iteration algorithm for $m = 1$ and so enjoys the same space-saving and parallelization benefits as its more generalized cousin.

If a simple approximation to the dominant eigenvalue of a matrix is desired, one could use the *Rayleigh quotient*. It is based on the idea that if x is an eigenvector of A corresponding to $\lambda \in \Lambda_A$ and if $y \stackrel{\text{def}}{=} x + O(\epsilon) \cdot u$ where $u = [1, 1, \cdots, 1]^{\mathrm{T}}$, then

$$\frac{y^{\mathrm{T}} A y}{y^{\mathrm{T}} y} = \lambda + O(\epsilon^2)$$

Once again, the crucial computation is a matrix-vector multiplication.

Lanczos [4] describes a method which provides a solution to the eigenvalue problem for symmetric, positive definite matrices. The idea is to find a positive definite matrix Q such that $Q^T A Q = T$, where T is tridiagonal. The diagonal element α_{ii} is given by

$$\alpha_{ii} = {q_i}^{\mathrm{T}} A q_i$$

which can be calculated in parallel much the same way we did in this thesis. Note that the matrix A need not be explicitly stored. The off-diagonal elements will tend toward zero, but the columns of Q may experience loss of orthogonality. Parlett [7] describes a method for performing a selective reorthogonalization of several of these columns.

REFERENCES

[1] Bauer, F. L. “Das Verfahren der Treppeniteration und verwandte Verfahren zur Lösung algebraischer Eigenwertprobleme.” *Z. angew. Math. Phys*. **8**.

[2] Dongarra, J. J., *et. al.* *LINPACK Users' Guide*. Philadelphia: SIAM, 1979.

[3] Golub, Gene H. and van Loan, Charles F. *Matrix Computations*. Baltimore: The Johns Hopkins University Press, 1983.

[4] Lanzcos, C. “An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators.” *Journal of Research of the National Bureau of Standards*, **45**:255-282, 1950.

[5] Noble, Ben and Daniel, James W. *Applied Linear Algebra*. 2nd ed. Englewood Cliffs: Prentice-Hall, Inc., 1977.

[6] O'Leary, D. P., Stewart, G. W., and van de Geijn, Robert. “DOMINO: A Message Passing Environment for Parallel Computation.” University of Maryland Department of Computer Science Technical Report TR-1648. College Park: April, 1986.

[7] Parlett, B. N. “The Lanzcos Algorithm with Selective Orthogonalization.” *Mathematics of Computation*, **22**:217-238, 1979.

[8] Smith, B. T., *et. al.* *Matrix Eigensystem Routines -- EISPACK Guide*. 2nd ed. New York: Springer-Verlag, 1976.

[9] Stewart, G. W. “Communication in Parallel Algorithms: An Example.” (unpublished).

[10] —. *Introduction to Matrix Computations*. New York: Academic Press, 1973.

[11] —. “Simultaneous Iteration for Computing Invariant Subspaces of Non-Hermitian Matrices.” *Numer. Math*. **25**.

[12] —. "SRRIT -- A FORTRAN Subroutine to Calculate the Dominant Invariant Subspaces of a Real Matrix." University of Maryland Department of Computer Science Technical Report TR-514. College Park: November, 1978.

[13] Wilkinson, J. H. *The Algebraic Eigenvalue Problem*. Oxford, England: Clarendon Press, 1965.