Chapter 9

Fast Linear Solvers

We have already discussed how to solve tridiagonal linear systems of equations using direct solvers (the Thomas algorithm) in chapter 6 and some iterative solvers (Jacobi, Gauss-Seidel, SOR and multigrid) in chapter 7. We have also discussed solutions of nonlinear and linear systems and have introduced the conjugate gradient method in chapter 4. In the current chapter we revisit this subject and present general algorithms for the *direct* and *iterative* solution of large linear systems. We start with the classical Gaussian elimination (which is a fast solver!) and then proceed with more sophisticated solvers and preconditioners for symmetric and non-symmetric systems.

In parallel computing, we introduce the broadcasting command *MPI_Bcast*, and demonstrate its usefulness in the context of Gaussian elimination. In addition, we reiterate the use of *MPI_Send*, *MPI_Recv*, *MPI_Allgather*, and *MPI_Allreduce* through example implementations of algorithms presented in this chapter.

9.1 Gaussian Elimination

Gaussian elimination is one of the most effective ways to solve the linear system

$$\mathbf{Ax} = \mathbf{b}.$$

The Thomas algorithm, see section 6.1.4, is a special case of Gaussian elimination for tridiagonal systems.

The computational complexity of Gaussian elimination is associated with the size and structure of the $n \times n$ matrix $\mathbf{A}$, and so is its accuracy. It is based on the "superposition principle" for linear systems, i.e., the fact that we can replace equations of the original system with equivalent equations formed as linear combinations of the rows of $\mathbf{A}$ and corresponding values of $\mathbf{b}$. In its simplest form it states:

- Take each row and subtract a multiple of it from subsequent rows in order to zero out the element of $\mathbf{A}$ below the diagonal.

We demonstrate this by the following example.

Example: Consider the 3×3 system

$$\boxed{x_1} \;+\; \frac{1}{2}x_2 + \frac{1}{3}x_3 = 3 \tag{9.1}$$

$$\boxed{\frac{1}{2}x_1} \;+\; \frac{1}{3}x_2 + \frac{1}{4}x_3 = 2 \tag{9.2}$$

$$\boxed{\frac{1}{3}x_1} \;+\; \frac{1}{4}x_2 + \frac{1}{5}x_3 = 1 \tag{9.3}$$

We will solve this (3×3) system in two stages of elimination.

Stage 1: In the first stage we target the first term placed in a box in the equation above. To this end, we select the *pivot* $a_{11} = 1$ and also the multipliers:

$$\ell_{21}^{(1)} = \frac{a_{21}}{a_{11}} = \frac{1/2}{1} = 1/2 \qquad \text{and} \qquad \ell_{31}^{(1)} = \frac{a_{31}}{a_{11}} = \frac{1}{3}\,.$$

We then multiply equation (9.1) by $\ell_{21}^{(1)}$ and subtract it from equation (9.2). We also multiply equation (9.1) by $\ell_{31}^{(1)}$ and subtract it from equation (9.3). The resulting two new equations replace (9.2) and (9.3), i.e.,

$$x_1 + \frac{1}{2}x_2 \;+\; \frac{1}{3}x_3 = 3$$

$$\boxed{\frac{1}{12}x_2} \quad + \quad \frac{1}{12}x_3 = 1/2 \tag{9.4}$$

$$\boxed{\frac{1}{12}x_2} \quad + \quad \frac{4}{45}x_3 = 0 \tag{9.5}$$

Stage 2: Next, we target the x_2 term, and choose a new pivot and new multipliers, respectively, as:

$$a_{22}^{(1)} = \frac{1}{12}; \qquad \ell_{32}^{(2)} = \frac{1/12}{1/12} = 1,$$

and proceed as before by multiplying equation (9.4) by $\ell_{32}^{(2)}$ and subtracting it from equation (9.5), we obtain

$$x_1 + \frac{1}{2}x_2 \quad + \quad \frac{1}{3}x_3 = 3$$

$$\frac{1}{12}x_2 \quad + \quad \frac{1}{12}x_3 = \frac{1}{2} \tag{9.6}$$

$$\frac{1}{180}x_3 = -\frac{1}{2} \tag{9.7}$$

We now see that the system of equations (9.1), (9.6) and (9.7) can be solved easily by back substitution starting from equation (9.7), then equation (9.6) and finally equation (9.1), to obtain:

$$\begin{aligned} x_3 &= -90 \\ x_2 &= 12\left[\frac{1}{2} - \frac{1}{12}(-90)\right] = 96 \\ x_1 &= 3 - \frac{1}{2}(96) - \frac{1}{3}(-90) = -15 \end{aligned}$$

In matrix form, the system of equation (9.7) is

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{12} & \frac{1}{12} \\ 0 & 0 & \frac{1}{180} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix}$$

so the coefficient matrix is *upper triangular*; we will denote this matrix by $\mathbf{U}$. We also collect all the multipliers $\ell_{ij}^{(k)}$ we have calculated to form the following *lower triangular* matrix $\mathbf{L}$

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{3} & 1 & 1 \end{bmatrix},$$

where we have places 1's in the diagonal. We can verify that

$$\mathbf{A} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{3} & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{12} & \frac{1}{12} \\ 0 & 0 & \frac{1}{180} \end{bmatrix} = \mathbf{L} \cdot \mathbf{U}$$

Therefore, the two first stages of Gaussian elimination resulted in the *factorization* of $\mathbf{A}$ into an $\mathbf{LU}$ product. Both $\mathbf{L}$ and $\mathbf{U}$ are special matrices, and this leads to substantial reduction in computational complexity.

Remark: The matrix $\mathbf{A}$ employed in this example is a special matrix that has elements $a_{ij} = \frac{1}{i+j-1}$ so the n^{th} row is the vector

$$\left(\frac{1}{n}, \frac{1}{n+1}, \frac{1}{n+2}, \dots, \frac{1}{2n-1}\right)^T.$$

For n large, e.g., $n = 1000$, the entries of the last row are about three orders of magnitude smaller than the entries of the first row. This large disparity leads to many difficulties because of the *ill-conditioning* of this matrix. This matrix was first introduced by the famous mathematician David Hilbert, and it is called the *Hilbert matrix*; it is an example of an ill-conditioned matrix. Its condition number is large, e.g., greater than 10^5 for $n \geq 5$.

9.1.1 LU Decomposition

We now generalize the Gaussian elimination procedure to an $n \times n$ system

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \ldots & + & a_{nn}x_n & = & b_n\,. \end{array}$$

In the general case we need $(n-1)$ stages of elimination in order to arrive at the upper triangular system. We will assume that all the pivots at every stage k are $a_{(ii)}^{(k)} \neq 0$, but we will remove this constraint later when we discuss algorithms that involve row and/or column pivoting.

The *first stage* of elimination leads to

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\ & & a_{22}^{(1)}x_2 & + & \ldots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\ & & \vdots & & & & \vdots & & \vdots \\ & & a_{n2}^{(1)}x_2 & + & \ldots & + & a_{nn}^{(1)}x_n & = & b_n^{(1)}\,. \end{array}$$

Here the intermediate coefficients $a_{ij}^{(1)}$ are defined by

$$a_{ij}^{(1)} = a_{ij} - a_{1j}\ell_{i1}^{(1)}; \quad \ell_{i1}^{(1)} = \frac{a_{i1}}{a_{11}}$$

and the entries on the right-hand-side are

$$b_i^{(1)} = b_i - b_1\frac{a_{i1}}{a_{11}}\,.$$

Similarly, the second stage of elimination produces $a_{ij}^{(2)}$ and $b_i^{(2)}$, and so on, until the $(n-1)^{th}$ stage, where we obtain $a_{nn}^{(n-1)}$ and $b_n^{(n-1)}$.

This procedure is the forward substitution and gives both matrices $\mathbf{L}$ and $\mathbf{U}$. In particular, we replace $\mathbf{Ax} = \mathbf{b}$ by

$$\mathbf{L}\underbrace{\mathbf{Ux}}_{\mathbf{y}} = \mathbf{b} \Rightarrow \mathbf{Ly} = \mathbf{b},$$

where

$$\mathbf{U}\mathbf{x} = \mathbf{y}\,.$$

We can now summarize the solution procedure, which consists of three main steps, as follows:

1. **LU** decomposition: $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$
2. Forward Solve for $\mathbf{y}$: $\mathbf{L}\mathbf{y} = \mathbf{b}$
3. Backward Solve for $\mathbf{x}$: $\mathbf{U}\mathbf{x} = \mathbf{y}$

The pseudo-code for steps (1) and (2) is:

```
for $k = 1, n-1$
    for $i = k+1, n$

    $\ell_{ik} = \frac{a_{ik}}{a_{kk}}$ (assuming $a_{kk} \neq 0$)

    for $j = k, n$
        $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
    endfor
    $b_i = b_i - \ell_{ik} b_k$
    endfor
endfor
```

Computational Cost: The *operation count* for the above code is obtained by considering first the innermost loop j, then the loop i, and finally adding operations from all elimination stages: $k = 1$ to $n-1$. Thus, we have

$$W_{LU} = 2\sum_{k=1}^{n-1}(n-k)\cdot(n-k) = 2\sum_{m=1}^{n-1} m^2 = 2\frac{(n-1)n(2n-1)}{6} \approx \frac{2}{3}n^3,$$

where the factor 2 accounts for one addition and one multiplication. If only multiplications are counted then

$$W_{LU} \approx \frac{n^3}{3}\,,$$

which is the operation count often quoted in the literature.

The *third step* in the solution is the *backward substitution*, which yields first

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

and marching backward all the way to the first entry, we have

$$x_1 = \frac{b_1 - a_{12}x_2 \ldots - a_{1n}x_n}{a_{11}}.$$

The following pseudo-code represents this back solve:

```
for $k = n, 1$ (*reverse ordering*)
    $x_k = b_k$ (*initialization*)
    for $i = k + 1, n$
        $x_k = x_k - a_{ki}x_i$
    endfor
    $x_k = x_k / a_{kk}$
endfor
```

We conclude from the above that the operation count for the backward/forward substitution is $\mathcal{O}(n^2)$.

Remark 1: In the pseudo-code above, steps (1) and (2) are accomplished together for computational efficiency. Solving for $\mathbf{y}$ amounts to adding the $(b_i = b_i - \ell_{ik}b_k)$ line in the appropriate place. At the conclusion of this algorithm, the matrix $\mathbf{A}$ has been over-written with the upper triangular matrix $\mathbf{U}$, and the vector $\mathbf{b}$ has been over-written with the solution of $\mathbf{Ly} = \mathbf{b}$. All that remains is to accomplish the backsolve for $\mathbf{Ux} = \mathbf{y}$.

Remark 2: Note that in both pseudo-codes above we have attempted to minimize the required memory by over-writing onto the same memory locations. However, this should be avoided in cases where we are interested in using the matrix $\mathbf{A}$ again somewhere else in our program. In the codes above both the entries of matrix $\mathbf{A}$ and the entries of the right-hand-side are over-written.

Remark 3: We have already discussed the *Thomas algorithm* in section 6.1.4, which is a subcase of the LU decomposition presented here with bandwidth $m = 1$. In general for a banded matrix with order n and (semi-) bandwidth m the operation count is

- LU decomposition: $\mathcal{O}(m^2 n), m \ll n$
- Back Solve: $\mathcal{O}(mn), m \ll n$

Remark 4: The Gram-Schmidt QR factorization of a matrix presented in chapter 2 produces an upper triangular matrix $\mathbf{R}$ but the $\mathbf{Q}$ matrix is an orthogonal *full* matrix. The Gram-Schmidt algorithm costs $\mathcal{O}(2n^3)$ (including addition and multiplications), i.e., it is *three times* more expensive than the LU algorithm. However, the QR decomposition can also be achieved by the *Householder method* which is only twice as expensive, i.e., it costs $\mathcal{O}(\frac{4}{3}n^3)$, see section 9.3 below.

Remark 5: (*Cramer versus Gauss*) We compare here the cost for solving $\mathbf{Ax} = \mathbf{b}$ using the Cramer method of determinants (time t_C) which is $\mathcal{O}(n!)$ versus the Gaussian elimination method (time t_G) which is $\mathcal{O}\left(\frac{2n^3}{3}\right)$. We assume that we use a processor with sustained speed of 1 Gflops[1].

$$n = 3: \quad \begin{aligned} t_C &\approx \frac{3!}{10^9} \text{ seconds } \approx 6 \textit{ nanoseconds} \\ t_G &\approx \frac{2}{3}\frac{3^3}{10^9} \text{ seconds } \approx 18 \textit{ nanoseconds} \end{aligned}$$

$$n = 10: \quad \begin{aligned} t_C &\approx \frac{10!}{10^9} \text{ seconds } \approx 3 \textit{ milliseconds} \\ t_G &\approx \frac{210^3}{3 \cdot 10^9} \text{ seconds } \approx 0.6 \textit{ microseconds} \end{aligned}$$

$$n = 20: \quad \begin{aligned} t_C &\approx \frac{10!}{10^9} \text{ seconds } \approx 675,806 \text{ h}\textit{ours} \approx 28,1585 \text{ days } \approx 80 \textit{ years} \\ t_G &\approx \frac{2}{3}\frac{20^3}{10^9} \text{ seconds } \approx 5 \textit{ microseconds} \end{aligned}$$

Clearly, Gauss wins by years! In figure 9.1 we plot the growth in computational work of Cramer's method versus the Gaussian elimination method.

Remark 6: The Gaussian elimination offers an efficient way of computing the determinant of $\mathbf{A}$, since

$$\begin{aligned} \det(\mathbf{A}) &= \det(\mathbf{L}) \det(\mathbf{U}) \\ &= 1 \cdot [u_{11} \cdot u_{22} \ldots u_{nn}] \end{aligned}$$

where u_{ii}, $i = 1, \ldots, n$ are the diagonal elements of the upper triangular matrix $\mathbf{U}$. We recall that the determinant of any triangular matrix is simply the product of its diagonal elements.

[1]The TeraHertz processor is already in the horizon; it will consist of about one billion transistors!

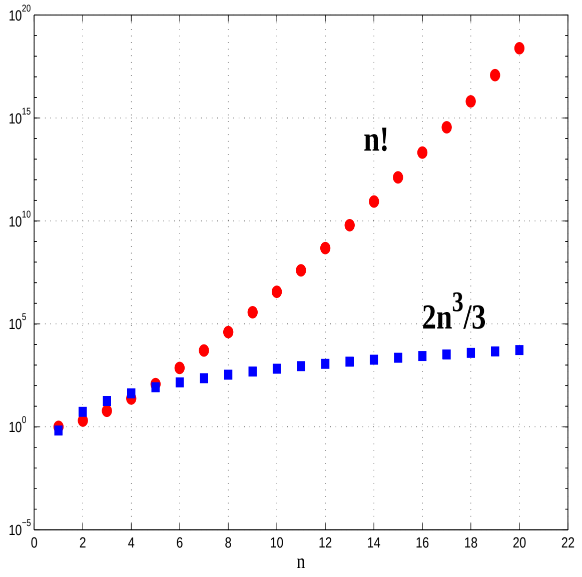

Figure 9.1: Comparison of the growth of computational work in Cramer's method, $n!$, versus Gaussian elimination method, $\frac{2}{3}n^3$.

Remark 7: The Gaussian elimination can be used to explicitly construct the inverse $\mathbf{A}^{-1}$ by setting the columns of the identity matrix $\mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$ as

$$\mathbf{b}_i = (0 \ldots \ 0 \ \underbrace{1}_{index\ i} \ \ 0 \ldots 0)^T$$

with only the i^{th} entry being non-zero, and solve

$$\mathbf{A}\mathbf{x}_i = \mathbf{b}_i, \ i = 1, \ldots, n.$$

The solution vector $\mathbf{x}_i$ forms the column i^{th} of the inverse $\mathbf{A}^{-1}$. Note that this involves only one LU decomposition and n back solves of $\mathcal{O}(n^2)$ and the total cost is *still* $\mathcal{O}(n^3)$.

9.1.2 To Pivot or Not to Pivot?

So far we have conveniently assumed that all the pivoting elements are non-zero, i.e.,

$$a_{ii}^{(k)} \neq 0$$

but this is not guaranteed for all problems! In practice, these pivots may be zero or very small numbers so that the multiplies $\ell_{ij}^{(k)}$ can potentially be very large numbers. To understand the effect of this, let us consider the 2×2 matrix

$$\mathbf{A} = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}$$

where $\epsilon \ll 1$. The condition number of $\mathbf{A}$ is $\kappa_2(\mathbf{A}) \to 2.6180$ as $\epsilon \to 0$ so this is a well conditioned matrix. We now obtain the LU decomposition of $\mathbf{A}$:

The $\mathbf{L}$ matrix is

$$\mathbf{L} = \begin{bmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{bmatrix}$$

and the $\mathbf{U}$ matrix is

$$\mathbf{U} = \begin{bmatrix} \epsilon & 1 \\ 0 & 1 - \epsilon^{-1} \end{bmatrix}.$$

For ϵ sufficiently small, $1 - \epsilon^{-1} \approx -\epsilon^{-1}$ and thus

$$\mathbf{U} \approx \begin{bmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{bmatrix},$$

so

$$\mathbf{L} \cdot \mathbf{U} = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix},$$

which is different than the original matrix $\mathbf{A}$ in the $(2, 2)$ entry, since

$$\mathbf{A} = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}.$$

We note that for any value of $a_{22} \neq 1$ (of order one) we get the same answer, which is obviously wrong! This is an example of a *numerical instability.* It is due to the fact that the condition number of $\mathbf{L}$ and $\mathbf{U}$ is extremely large unlike the condition number of the matrix $\mathbf{A}$ which is order one. This problem can be avoided if we simply reverse the order of the equations, i.e., interchange the rows, and work with the re-ordered matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix}$$

as now the multiplier is $\epsilon < 1$.

We can generalize the result above and apply row interchange, which is also called *partial pivoting*, to obtain multipliers

$$|\ell_{ij}| < 1\,.$$

The following pseudo-code describes Gaussian elimination with partial pivoting. It is an extension of the code we described earlier in the forward solve.

```
for $k = 1, n-1$
    $|a_{mk}| = \max\{|a_{kk}|, |a_{k+1k}|, \ldots |a_{nk}|\}$
    $p = m$
    for $q = k,\ n$
        $c = a_{kq}$
        $a_{kq} = a_{pq}$
        $a_{pq} = c$
    endfor
    for $i = k+1, n$
        $\ell_{ik} = a_{ik}/a_{kk}$
        for $j = k+1,\ n$
            $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$
        endfor
    endfor
endfor
```

We note that for the system $\mathbf{Ax} = \mathbf{b}$ we also need to interchange appropriately the right-hand-side. Unlike the standard Gauss elimination, in the partial pivoting case we have

$$\mathbf{A} \neq \mathbf{LU}$$

but $\mathbf{A} = \mathbf{P}^{-1}\mathbf{LU}$ is true, where $\mathbf{P}$ is a permutation matrix describing the partial pivoting.

While partial (row-pivoting) works effectively in practice, there are a few pathological cases where even this may breakdown. In these cases, we can perform an additional similar pivoting by columns, searching for a maximum pivot along both rows and columns. In general, indications of an ill-conditioned matrix are provided by the small magnitude of the pivot or the large magnitude of the solution compared to the right-hand-side, although there are matrices which do not have these properties but they are still ill-conditioned.

Remark 1: For bounded matrices, (e.g., the tridiagonal systems involved in Thomas algorithms; see section 6.1.4), partial pivoting and complete pivoting result in increasing the bandwidth and even producing *full* matrices. Therefore, the computational work instead of being linear as in the Thomas algorithm may become $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$ for row or row/column pivoting, respectively.

Software Suite

Putting it into Practice

The function below is an implementation of both the pivot and non-pivot versions of Gaussian elimination by LU decomposition. Three arguments are required: the matrix **A**, the vector **b**, and an integer parameter *pivotflag* denoting whether the pivoting should be enabled (zero for no-pivoting and one for pivoting).

```
void GaussElimination(SCMatrix &A, SCVector &b, int pivotflag){
  int pivot;
  int N = A.Rows();

  /* NOTE: The values contained in both the matrix A and
     the vector b are modified in this routine.  Upon
     returning, A contains the upper triangular matrix
     obtained from its LU decomposition, and b contains
     the solution of the system Ax=b*/

  // Steps (1) and (2) (decomposition and solution of Ly = b)
  switch(pivotflag){
  case 1: // Case in which pivoting is employed

    for(int k=0; k < N-1; k++){
      pivot = A.MaxModInColumnindex(k,k);
      A.RowSwap(pivot,k);
      Swap(b(pivot),b(k));
      for(int i=k+1;i<N;i++){
        double l_ik = A(i,k)/A(k,k);
        for(int j=k;j<N;j++)
          A(i,j) = A(i,j) - l_ik*A(k,j);
        b(i) = b(i) - l_ik*b(k);
      }
    }
    break;
  case 0:  // Case 0/default in which no pivoting is used
  default:

    for(int k=0; k < N-1; k++){
      for(int i=k+1;i<N;i++){
        double l_ik = A(i,k)/A(k,k);
        for(int j=k;j<N;j++)
          A(i,j) = A(i,j) - l_ik*A(k,j);
        b(i) = b(i) - l_ik*b(k);
      }
    }
  }

  // Step (3) (backsolving to solve Ux=y)
  b(N-1) = b(N-1)/A(N-1,N-1);
  for(int k=N-2;k>=0;k--){
    for(int j=k+1;j<N;j++)
      b(k) -= A(k,j)*b(j);
    b(k) = b(k)/A(k,k);
  }
}
```

Remark 2: In the implementation above we use a *switch* statement to partition the different cases. Two advantages are: code readability and the ease of adding another condition. Suppose that we decided to implement a new pivoting algorithm; within the current function we could merely add another case to denote the new pivoting functionality.

Notice that it is not required that case '0' go before case '1'. Recall that the switch statement evaluates the validity of the switch in the order given in the code, hence in this case the switch statement first checks to see if case '1' is valid, and if not checks for case '0'. Also notice that we use a 'default' case. If neither a *zero* or a *one* is given in the switch input, the default will be executed. In this example, the default case and case '0' are identical.

Remark 3: In the code above, we utilize two *Matrix class* methods:

1. Matrix::MaxModInColumnindex(...), and
2. Matrix::Rowswap(...).

The first method implements the maximum modulus in a column operation needed for pivoting. The second method swaps two rows of the matrix A. The advantage of using these two methods is that from this level in the code the implementation details of how these two functions are accomplished are not our concern. At the level of this code we merely need to know the input, output, and contract for the methods.

Remark 4: Notice that in the code above we both declare and initialize the iterating value (such as `int k=0`) within the *for* loops. Recall that C++ does not require us to declare all variables at the beginning of the function. In this function we have made liberal use of this ability by declaring and initializing each iterating value with its respective *for* loop.

The need for pivoting is dictated by the properties of the matrix **A**. There are two special categories of matrices for which we *do not need* to pivot. These matrices are:

- Diagonally-dominant, or
- Positive-definite.

A *strictly diagonally-dominant* matrix has the property

$$|a_{ii}| > \sum_{j=1}^{n} |a_{ij}|, \ i = 1, \ldots, n; \quad j \neq i$$

and it is guaranteed to be non-singular.

A *positive-definite* matrix is defined by the condition

$$\forall \mathbf{x} \neq 0, \ \ \mathbf{x}^T \mathbf{A} \mathbf{x} > 0,$$

which guarantees that the matrix **A** is non-singular. In addition, the above properties guarantee that there will be no numerical instabilities in systems where such matrices are involved.

Fortunately, many of the algebraic systems resulting from PDEs that describe physical phenomena (presented in chapters 6 and 7) have these desirable properties. In particular, the matrix obtained from the discretization of d^2u/dx^2 (see chapter 6) is tridiagonal with diagonals $(1, -2, 1)$. Although it does not satisfy strictly the diagonal dominance condition, it is an *irreducible matrix*, i.e., its associated directed graph is strongly connected, and this condition is equivalent to diagonal dominance.

The special matrices we have described are guaranteed to be non-singular but they are also far from being even approximately singular. Clearly, if a matrix $\mathbf{A}$ is singular, then Gaussian elimination cannot be applied as $\mathbf{A}^{-1}$ does not exist. However, in practice many matrices are *almost singular*, and this is the condition we should investigate, as it leads to numerical instabilities. The value of the determinant, if the matrix is scaled properly, can give us an indication if the matrix is almost singular or *ill-conditioned.* However, computing determinants is costly and at least equivalent in computational complexity to an LU decomposition, the stability of which we investigate in first place! Only for special matrices the computation of determinant may be employed.

Another approach is to employ the condition number $\kappa(\mathbf{A})$ for matrix $\mathbf{A}$, i.e.,

$$\kappa(\mathbf{A}) \equiv \parallel \mathbf{A} \parallel \cdot \parallel \mathbf{A}^{-1} \parallel \ .$$

As we have seen in chapter 2 it relates the perturbation of data to the changes in the solution, i.e.,

$$\frac{\parallel A\mathbf{x} \parallel}{\parallel x \parallel} \leq \kappa(\mathbf{A}) \frac{\parallel A\mathbf{b} \parallel}{\parallel b \parallel} .$$

The condition number is always computed with respect to some norm. We will use the notation $\kappa_i(\mathbf{A})$ to denote the condition number of $\mathbf{A}$ with respect to the $\parallel \cdot \parallel_i$ norm. When no subscript is given, we assume that the $\parallel \cdot \parallel_2$ norm is used. We can define relationships between different norm-based condition numbers using the equivalence relations between the norms.

By definition the condition number $\kappa(\mathbf{A})$ is greater or equal to one, but we are interested in extremely large values of $\kappa(\mathbf{A})$, as it acts as an amplifier in the propagation of disturbance (noise) from the input to output (solution).

For symmetric matrices we have that

$$\kappa_2(\mathbf{A}) = \frac{|\lambda_{\max}|}{|\lambda_{\min}|} ,$$

and we have computed the eigenvalues λ_i for several cases in chapter 6; see also chapter 10. For general matrices, however, it is difficult to compute the condition number economically so approximate estimation algorithms are employed. Here we will present the estimator proposed by Hager [52]; see also [26]. The algorithm[2] obtains a lower bound on the inverse matrix in the one-norm. However, we note that $\mathbf{A}^{-1}$ is not constructed explicitly as this is costly, in fact

[2]This algorithm is available in LAPACK, routines **sgesvx** and **slacon**.

it requires an $\mathcal{O}(n^3)$ operation! Instead, the matrix-vector products $\mathbf{A}^{-1}\mathbf{x}$ are computed on-the-fly:

```
Initialize:   $\mathbf{x} : \| \mathbf{x} \|_1 = 1$

Begin Loop :  $\mathbf{y} = \mathbf{A}^{-1}\mathbf{x}$
              $\mathbf{z} = sign(\mathbf{y})$
              $\mathbf{q} = (\mathbf{A}^{-1})^T\mathbf{z}$
              if $\| \mathbf{q} \|_\infty \leq \mathbf{q}^T\mathbf{x}$ return $\| \mathbf{y} \|_1$
                  elseif $\mathbf{x} = \mathbf{e}_j sign(q_j)$ with $|q_j| = \| \mathbf{q} \|_\infty$
              endif
End Loop
```

Here $\mathbf{e}_j = (0, 0, \ldots, 1, \ldots 0)^T$ is the j^{th} column of the identity matrix. The condition number is then estimated from

$$\kappa \approx \| \mathbf{A} \|_1 \cdot \| y \|_1 \, .$$

We note that $\| \mathbf{y} \|_1$ is a local maximum to $\| \mathbf{A}^{-1}\mathbf{x} \|_1$, and that this method is based on computing the gradient of $f(x) \approx \| \mathbf{A}^{-1}\mathbf{x} \|_1$; for a detailed explanation of the algorithm see [52].

Remark 5: The accuracy degradation of the solution, if the condition number is large, can be estimated as follows: Assuming that the condition number is $\kappa(\mathbf{A}) \approx 10^p$, then the number of accurate digits in the solution is $(q - p)$ if the solution is computed in *q-digit* arithmetic.

9.1.3 Parallel LU Decomposition

The efficient parallel implementation of decomposing a non-singular matrix $\mathbf{A}$ into its LU factorization requires that we address two main issues:

1. How to split the matrix $\mathbf{A}$ among the processors.
2. How to organize the triple loop so that efficient BLAS operations can be employed.

We consider here distributed memory computers so only parts of the matrix $\mathbf{A}$ are stored in each processor. The obvious ways to split the matrix $\mathbf{A}$ are by rows or by columns, but it may also be beneficial to split it in blocks. A better layout, often used in practice, is *interleaved storage* either by row or by column. We examine these different cases in some detail in the following.

With regards to organizing the triple nested loop – recall that we deal with an $\mathcal{O}(n^3)$ operation – there are six ways of permutating the indices (ijk), just like in matrix-matrix multiply that we discussed in chapter 2. In table 9.1 we present all six versions and basic operations involved.

We have defined the basic operation in table 9.1 as the operation involved in the innermost loop. In four versions this is a *daxpy* operation (i.e., double a (scalar) $\mathbf{x}$ (vector) plus $\mathbf{y}$ (vector), while in the other two versions (ijk and jik) it is a *ddot* (i.e., double dot product). These are both BLAS1 operations and cannot easily take advantage of *cache blocking* or *data re-use*, see section 2.2.6. Thus, appropriate modifications of these basic loops are required to be able to employ BLAS2 and BLAS3 routines. This will depend on the specific way we layout the matrix $\mathbf{A}$ in order to also maximize parallel efficiency. We examine this issue in more detail next.

Access By Rows

Let us first assume that the matrix $\mathbf{A}$ is accessed by rows as in the (kij) loop of table 9.1; see figure 9.2 for a schematic explanation. Let us also assume that the first processor P_1 holds the first row $\mathbf{a}_1^T$, P_2 holds $\mathbf{a}_2^T$ and so on.

During the *first* elimination stage, the processor P_1 needs to send its row to all other processors so that processors $P_2 \ldots P_n$ will simultaneously update their columns. Therefore, the operations

$$\left.\begin{array}{l} \ell_{i1} = \frac{a_{i1}}{a_{11}} \\ a_{ij} = a_{ij} - \ell_{i1} a_{1j} \end{array}\right\} \begin{array}{l} j = 2, \ldots n \\ P_2, \ldots P_n \end{array}$$

can be performed *in parallel.* During this first stage, processor P_1 remains essentially idle after it communicates with the rest of the processor.

The *second* stage also starts with a communication step as P_2 needs to broadcast its new row to all other processors $P_3 \ldots P_n$. It too remains idle after that, while $P_3 \ldots P_n$ update their rows in parallel, and so on for the remaining stages. The computations of the multiplies as well as the updates are done in parallel but after the k^{th} stage, k processors $(P_1 \ldots P_k)$ remain idle. This approach reduces significantly the parallel efficiency.

Some improvements can be made by *overlapping* communication with computation. For example, the broadcasting of the row of processor P_k can be done immediately after it is computed, so that the other processors receive it while they are updating their rows during the k^{th} stage of elimination. This overlapping of computation and communication is called *send-ahead* operation and it is quite common in distributed memory parallel computers. Another improvement would come about if we manage to increase the work done in parallel

1. ijk Loop: **A** - by column (**ddot**)

```
for $i = 2, n$
    for  $j = 2, i$
        $\ell_{i,j-1} = a_{i,j-1}/a_{j-1,j-1}$
        for $k = 1, j-1$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
    for  $j = i+1, n$
        for $k = 1, i-1$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
endfor
```

2. ikj Loop: **A**- by row (**daxpy**)

```
for  $i = 2, n$
    for  $k = 1, i-1$
        $\ell_{ik} = a_{ik}/a_{kk}$
        for $j = k+1, n$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
endfor
```

3. jik Loop: **A** - by column (**ddot**)

```
for  $j = 2, n$
    for  $p = j, n$
        $\ell_{p,j-1} = a_{p,j-1}/a_{j-1,j-1}$
    endfor
    for  $i = 2, j$
        for $k = 1, i-1$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
    for  $i = j+1, n$
        for  $k = 1, j-1$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
endfor
```

4. jki Loop: **A** - by column (**daxpy**)

```
for  $j = 2, n$
    for  $p = j, n$
        $\ell_{p,j-1} = a_{p,j-1}/a_{j-1,j-1}$
    endfor
    for  $k = 1, j-1$
        for  $i = k+1, n$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
endfor
```

5. kij Loop: **A** - by row (**daxpy**)

```
for  $k = 1, n-1$
    for  $i = k+1, n$
        $\ell_{ik} = a_{ik}/a_{kk}$
        for  $j = k+1, n$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
endfor
```

6. kji Loop: **A** - by column (**daxpy**)

```
for  $k = 1, n-1$
    for  $p = k+1, n$
        $\ell_{pk} = a_{pk}/a_{kk}$
    endfor
    for  $j = k+1, n$
        for $i = k+1, n$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
endfor
```

Table 9.1: Six different ways of writing the LU triple loops.

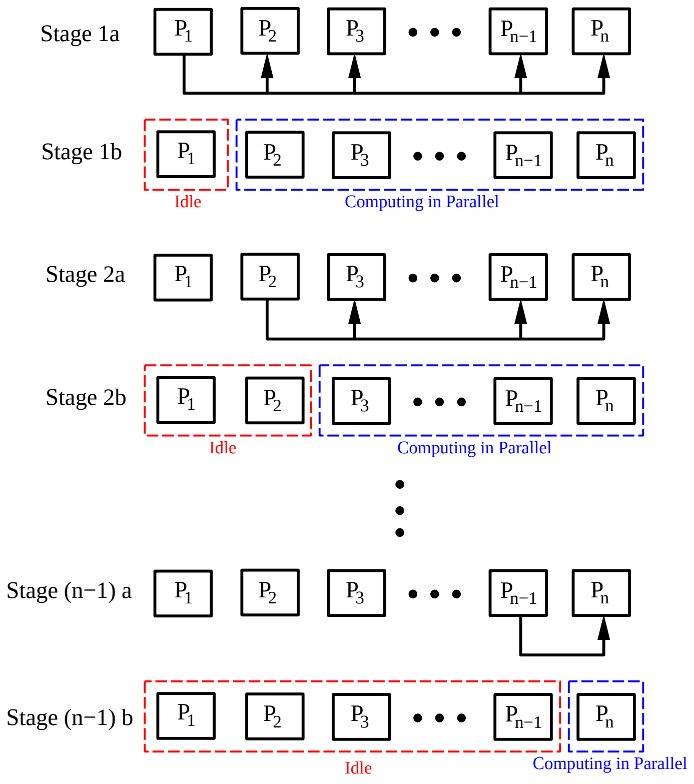

Figure 9.2: Schematic of communication (a) and computation (b) pattern when a matrix is partitioned such that each processor contains one row.

relative to communications. In general, the parallel efficiency improves as the ratio of computations to communications becomes larger. Thus, for given required communication the local work is increased as many rows are stored in each processor, so we have one, say, send-ahead operation every 100 rows on a $P = 10$ processor system and a matrix of $n = 1000$.

Blocked Layout

The row-blocked layout of the matrix $\mathbf{A}$ suggests that the practical way of storing $\mathbf{A}$ is by rows, and it extends also to a column-blocked layout, e.g., in the kji algorithm described in table 9.1. In this case, the multiplies ℓ_{ij} are computed on the processor P_j (or its corresponding number for blocked storage), and then ℓ_{ij} are broadcasted to all other (active) processors.

Partial pivoting in either the row-blocked or column-blocked parallel versions is handled differently. In the latter version, a search for the maximum pivot is performed within the processor while in the former (*kij* loop) the searching is across processors; this requires a fan-in algorithm for the *max* operator, as discussed in chapter 2.

Another improvement of the row-blocked or column-blocked layout schemes is to introduce a *cyclic* or *interleaved* layout, see figure 9.3. For example, assuming that we have available $k = 10$ processors for a 100×100 matrix $\mathbf{A}$, then we pursue the following storage scheme:

$$\begin{array}{lll} P_1\text{:} & \text{rows} & 1, 11, 21, \dots, 91 \\ P_2\text{:} & \text{rows} & 2, 12, 22, \dots, 92 \\ \vdots & & \\ P_{10}\text{:} & \text{rows} & 10, 20, \dots, 100 \end{array}$$

and similarly in a column-oriented storage scheme. Examining figure 9.3, we observe that during stage 1a processor one communicates row one to all other processors, and then in stage 1b all processors are active accomplishing row reduction. In comparison to the previous block setup in which rows one through ten would be assigned to processor one, rows 11 through 20 to processor two, etc., row two is now located on processor two. Hence in stage 2a, processor two communicates row two to all other processors, and in stage 2b all processors accomplish row reduction concurrently. In this manner, for the first $m = 90$ stages all the processors remain active both in communication and computation. Some inefficiencies may occur during the last few stages, however, as the final row assigned to a processor is eliminated, hence retiring the processor from service for the elimination.

An even better option is to employ a block of rows (or columns) and assign these blocks in a cyclic or interleaved manner. The advantages of this approach are that no processor retires early and that BLAS2 and BLAS3 routines can be used because of the blocking. All processors see roughly the same amount of work, proportioned to $1/P$, although the first processors work less, e.g., after they compute their first block.

Finally, a combination of row and column block interleaved storage of matrix $\mathbf{A}$ can be pursued. In this case we can imagine a mapping of b size blocks of $\mathbf{A}$ in a cyclic manner (of cycle length C) onto a *mesh-type* parallel computer consisting of $P = P_r \times P_c$ processors. Schematically, this arrangement is shown in figure 9.4.

This mapping can be established by setting

$$S_i = F(i, b) \text{ and } S_j = F(j, b),$$

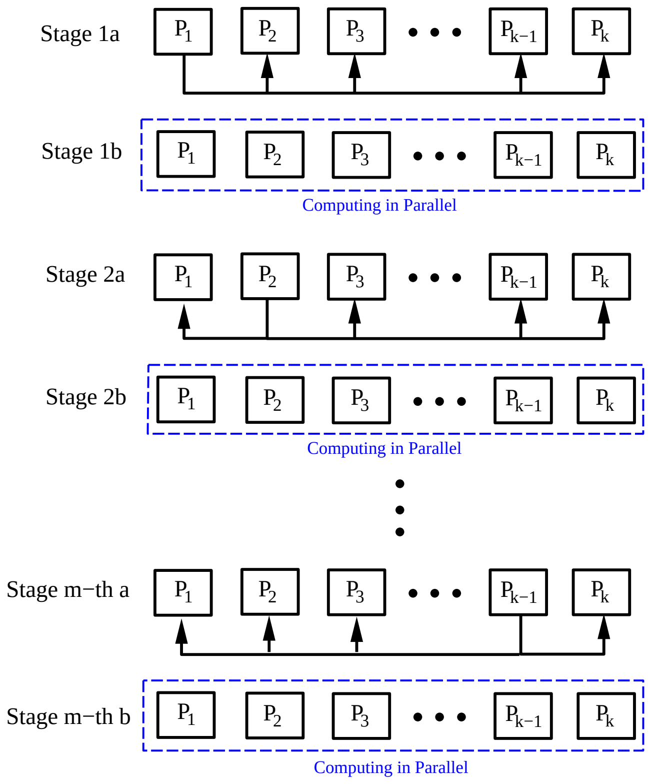

Figure 9.3: Schematic of communication (a) and computation (b) pattern when a matrix is partitioned with an interleaved layout.

with $i = 0, \ldots, n-1$ and $j = 0, \ldots, n-1$ (N being the size of $\mathbf{A}$) where the value of the (S_i, S_j) pair defines the processor in the mesh architecture. The function F is defined as

```
function $F(i, b)$
    floor $(i/b)$ modulo $C$
return;
```

where we assume that we deal with a square matrix and that $C^2 = P$. In the example of figure 9.4, we have $P = 4$ and $C = 2$. We have also assumed here for simplicity that the computer (mesh array) is symmetric but that may not be advantageous in practice, i.e, we may want to have $P_r > P_c$, that is more rows than columns. The advantage of this approach, in addition to its good parallel performance, is that it can make use of the BLAS3 routines, which

1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4

Figure 9.4: 2D block interleaved mapping of a 16×16 matrix on $P = 4$ processors; here $b = 2$ and $C = 2$, and the processor id used above is given by $S_i + 2S_j + 1$. Here S_i and S_j have the values 0 or 1.

provide high efficiency. The main steps of computing pivots, send-ahead, and parallel work on each processor that we presented earlier are also utilized here.

The plot of figure 9.5 shows a graphical representation of LU decomposition using BLAS3. The new entries to be computed are in the shaded area in the right lower corner of the matrix $\mathbf{A}$. The submatrix $\mathbf{A}_s$ that is currently shown to be computed has a size $b \times b$, i.e., the size of the block of rows and columns. This implementation is included in ScaLAPACK software for distributed computers, see

```
www.netlib.org/scalapack
```

9.1.4 Parallel Back Substitution

We first present an *ideal* algorithm for parallel back substitution that provides a lower bound for its computational complexity. Although the lower bound may be unattainable given hardware constraints, it provides us with the "best-case" scenario that the algorithm can provide given unrestricted resources. By understanding the concept which yields the lower bound result, we hope that after incorporating relevant constraints we will obtain a reasonable algorithm. The ideal algorithm for parallel back substitution is based on a *divide-and-conquer* algorithm for inverting triangular matrices.

Let us consider the lower triangular matrix $\mathbf{L}$, which we decompose into submatrices of half size $\mathbf{L}_1$ $\mathbf{L}_2$ and $\mathbf{L}_3$ as follows

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 & 0 \\ \mathbf{L}_2 & \mathbf{L}_3 \end{bmatrix}.$$

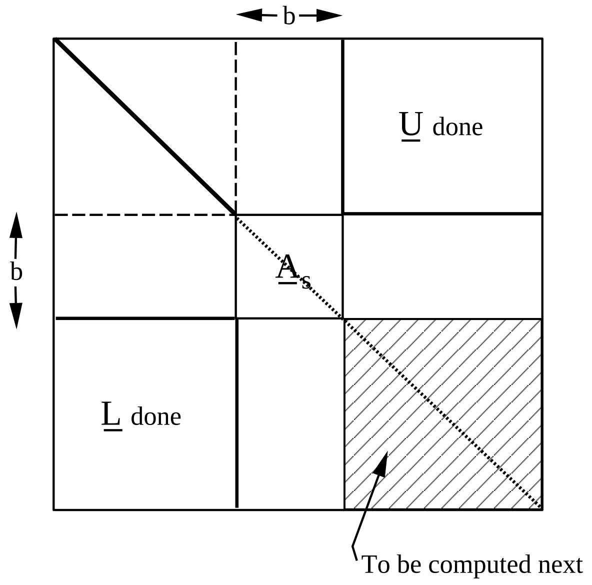

Figure 9.5: LU decomposition of a matrix **A** using 2D block interleaved storage. BLAS2 and BLAS3 can be effectively employed in this algorithm.

We assume that $n = 2^k$ and that k is an integer. This is shown schematically in the plot of figure 9.6.

It is clear that $\mathbf{L}_1$ and $\mathbf{L}_2$ are themselves lower triangular matrices. We can prove that

$$\mathbf{L}^{-1} = \begin{bmatrix} \mathbf{L}_1^{-1} & 0 \\ -\mathbf{L}_3^{-1}\mathbf{L}_2\mathbf{L}_1^{-1} & \mathbf{L}_3^{-1} \end{bmatrix},$$

and take advantage of this equation to set up a divide-and-conquer algorithm for inverting the matrix $\mathbf{L}$ of size n.

The main steps are shown in the following pseudo-code:

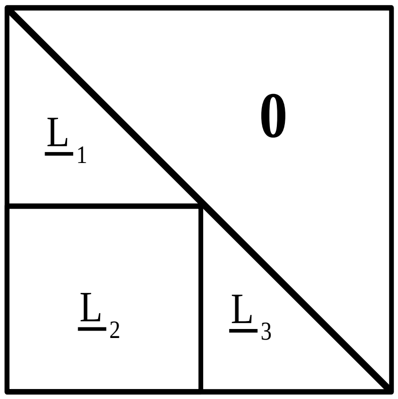

Figure 9.6: Decomposition of a lower triangular matrix $\mathbf{L}$. Matrices $\mathbf{L}_1$ and $\mathbf{L}_3$ are also lower triangular.

```
Function InvTriangular(L)
    If size(L)= 1 return 1/L
    Else
        Set L1 top triangular part of L
        Set L2 square part of L
        Set L3 bottom triangular part of L

        InvL1 =InvTriangular(L1)
        InvL3 =InvTriangular(L3)
        UpdateL2 = - InvL3 * L2 * InvL1

        return $L = \begin{vmatrix} InvL1 & 0 \\ UpdateL2 & InvL3 \end{vmatrix}$
    Endif
```

We can perform the inversion of $\mathbf{L}_1$ and $\mathbf{L}_2$ in parallel, so the cost C is

$$C[\text{InvTriangular}(n)] = C[\text{InvTriangular}(n/2) + C[mxm(n/2)],$$

where by mxm we denote the matrix-multiply. The ideal time for this matrix-multiply (assuming we employ n^3 processors) is $\mathcal{O}(\log n)$, and thus the ideal cost for inverting triangular matrices is $\mathcal{O}(\log^2 n)$. Unfortunately, this cost is impossible to realize in practice.

The $\mathcal{O}(\log^2 n)$ time estimate is based on the equation

$$t(n) = t(n/2) + \mathcal{O}(\log n).$$

We set $k = \log_2 n$, and thus

$$\begin{aligned} t(k) &= t(k-1) + \mathcal{O}(k) \\ &= t(k-2) + \mathcal{O}(k) + \mathcal{O}(k-1) \\ &= t(1) + \mathcal{O}(k) + \mathcal{O}(k-1) + \ldots + 1 \\ &\approx \frac{k(k-1)}{2} \sim \mathcal{O}(k^2/2) \sim \mathcal{O}(\log^2 n)\,. \end{aligned}$$

Note that the $\mathcal{O}(\log n)$ ideal estimate for matrix multiply is based on the fact that all entries of the product matrix can be computed in parallel on $P = n^3$ processors, and then summed up using a fan-in algorithm which is $\mathcal{O}(\log n)$. This is of course unattainable at the moment since the $n = 1000$ size (ideally parallel) matrix-multiply would require more than one billion processors, more than what we currently have on this planet!

We now discuss how the *back substitution* step $\mathbf{Ux} = \mathbf{y}$, which is also a triangular system, can be performed in parallel. The algorithm proceeds by computing first

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

while all other unknowns are computed from

$$\begin{aligned} \forall\, i &= n-1, \ldots 1 : \\ x_i &= \frac{b_i^{(i-1)} - a_{ii}^{(i-1)} x_{i+1}}{a_{i,i}^{i-1}} = \frac{b_i - u_{i,i+1}x_{i+1} - u_{i,i+2}x_{i+2} - \ldots u_{in}x_{in}}{u_{ii}}\,, \end{aligned}$$

where u_{ij} denote entries of the matrix $\mathbf{U}$.

The most obvious implementation of this is the following:

```
for $j = n, 1$
    for $j = i + 1, n$
        $b_i = b_i - u_{ij} x_j$
    endfor
    $x_i = b_i / u_{ii}$
endfor
```

This assumes that $\mathbf{U}$ is stored by rows, which are accessed in the innermost loop j. More specifically, the matrix $\mathbf{U}$ is stored in blocks of rows assigned to each processor. The corresponding operation employs the BLAS1 *ddot* routine, which implements the inner (dot) product of the innermost loop. However, if **U** is stored in blocks of columns per processor, then the (ij) loop of the *ddot* product implementation above has to be reversed as follows

```
for $j = n, 1$
   $x_j = b_j / u_{jj}$
   for $i = 1, j-1$
      $b_i = b_i - u_{ij} x_j$
   endfor
endfor
```

Here the innermost loop (i) generates updates of the numerators for the final answer x_j while the outer loop (j) sweeps the matrix **U** by columns starting from right to left. This is sometimes referred to as *left-looking* access of **U**, and it involves *daxpy* operations in the innermost loop.

Software Suite — *Putting it into Practice*

Prior to presenting a parallel implementation of Gaussian elimination, we will first discuss one new MPI function not introduced previously: *MPI_Bcast*. This function allows us to distribute to all processes within the communicator an identical piece of data. In the case of Gaussian elimination, *MPI_Bcast* will allow us to "broadcast" to all processes a particular row being used in the elimination. We will now present the function call syntax, argument list explanation, usage example and some remarks.

MPI_Bcast:

Function Call Syntax

```
int MPI_Bcast(
          void*          buffer     /*  in/out  */,
          int            count      /*  in      */,
          MPI_Datatype   datatype   /*  in      */,
          int            root       /*  in      */,
          MPI_Comm       comm       /*  in      */)
```

Understanding the Argument List

- *buffer* - starting address of the send buffer.
- *count* - number of elements in the send buffer.
- *datatype* - data type of the elements in the send buffer.
- *root* - rank of the process broadcasting its data.
- *comm* - communicator.

Example of Usage

```
int mynode, totalnodes;
int datasize; // number of data units to be broadcast
int root;     // process which is broadcasting its data

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

// Determine datasize and root

double * databuffer = new double[datasize];

// Fill in databuffer array with data to be broadcast

MPI_Bcast(databuffer,datasize,MPI_DOUBLE,root,MPI_COMM_WORLD);

// At this point, every process has received into the
// databuffer array the data from process root
```

Remarks

- Each process will make an identical call of the *MPI_Bcast* function. On the broadcasting (root) process, the *buffer* array contains the data to be broadcast. At the conclusion of the call, all processes have obtained a copy of the contents of the *buffer* array from process root, see figure 9.7.

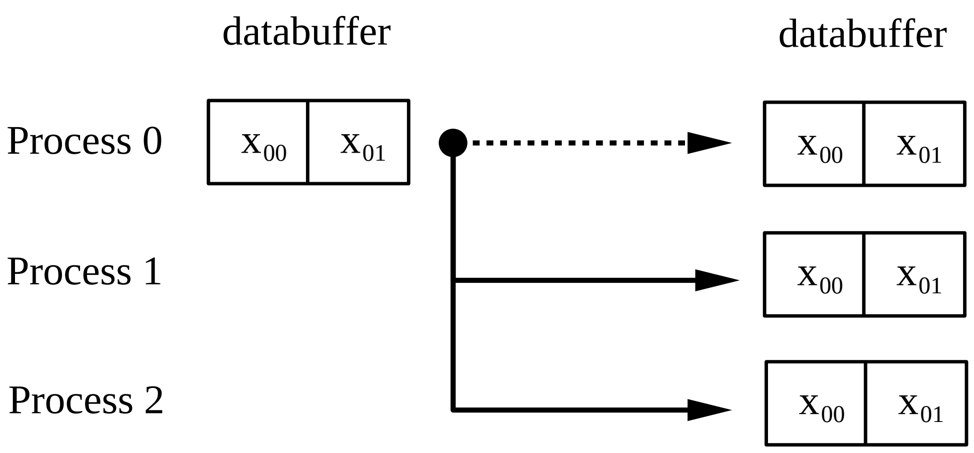

Figure 9.7: *MPI_Bcast* schematic demonstrating a broadcast of two data objects from process zero to all other processes.

We now present a parallel implementation of Gaussian elimination with back substitution. As a model problem, we solve for the interpolating polynomial of the Runge function (see section 3.1.4) by forming a Vandermonde matrix based on the Chebyshev points. Recall that the goal is to find the polynomial coefficients by solving the system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is the Vandermonde matrix and $\mathbf{b}$ is the function of interest evaluated at the interpolation points.

To better explain the code, we have broken the entire program into six parts, labeled part one through part six. The six parts break down the code as follows:

1. Part 1 - MPI initialization/setup and initial memory allocations.
2. Part 2 - Generation of the matrix rows local to each process.
3. Part 3 - Gaussian elimination of the augmented matrix.
4. Part 4 - Preparation for back substitution.
5. Part 5 - Back substitution to find the solution.
6. Part 6 - Program finalization and clean-up.

For each part, we will first present the code and then present a collection of remarks elucidating the salient points within each part.

Part 1 - MPI initialization

```
#include <iostream.h>
#include <iomanip.h>
#include "SCmathlib.h"
#include "SCchapter3.h"
#include<mpi.h>

void ChebyVandermonde(int npts, double *A, int row);

// Global variable to set size of the system
const int size = 10;

int main(int argc, char *argv[]){
  int i,j,k,index;
  int mynode, totalnodes;
  double scaling;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  int numrows = size/totalnodes;
  double **A_local = new double*[numrows];
  int * myrows = new int[numrows];
```

Remark 1: Notice that for this program we use a global constant variable to denote the size of the matrix system. By placing the variable declaration outside of the main function, the declaration is global to all functions (including the main function).

Remark 2: We have made the assumption that the size of the matrix is evenly divisible by the number of processors we are using. If this were not the case, we would have to properly take this into account by having different numbers of rows per processor.

$\boxed{\text{Part 2 - Generation of matrix rows}}$

```
  /* PART 2 */
  double * xpts = new double[size];
  ChebyshevPoints(size,xpts);
```

```
for(i=0;i<numrows;i++){
  A_local[i] = new double[size+1];
  index = mynode + totalnodes*i;
  myrows[i] = index;
  ChebyVandermonde(size,A_local[i],index);

  // Set-up right-hand-side as the Runge function
  A_local[i][size] = 1.0/(1.0+25.0*xpts[index]*xpts[index]);
}
delete[] xpts;


double * tmp = new double[size+1];
double * x = new double[size];
```

Remark 1: Notice that we allocate for each row $(size + 1)$ columns. Recall that in Gaussian elimination it is necessary for us to act on the right-hand-side (the vector **b**) as we do the row reduction. We can eliminate the extra communication cost that would come by handling the right-hand-side separately by forming an augmented matrix as shown in figure 9.8.

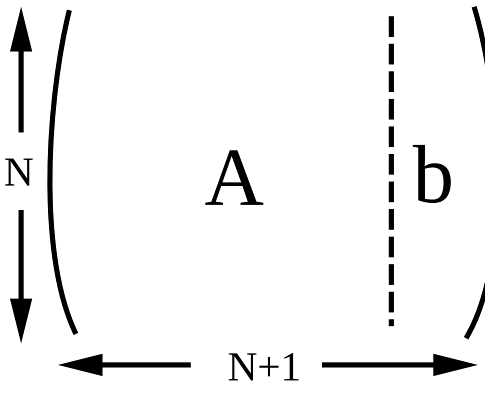

Figure 9.8: The augmented matrix consists of the original matrix **A** with the right-hand-side vector **b** appended as an additional column.

As you will see in the next section, all row reduction steps will be accomplished on $(size + 1)$ columns so that both the matrix and right-hand-side are updated properly.

Remark 2: To keep track of which rows a processor possesses, we store the indices of the rows in an array named *myrows*. This array contains *numrows* entries, each entry denoting for which rows in the matrix the processor is responsible.

Part 3 - Gaussian elimination

```
/* PART 3 */
/* Gaussian Elimination of the augmented matrix */
int cnt = 0;
for(i=0;i<size-1;i++){
  if(i == myrows[cnt]){
    MPI_Bcast(A_local[cnt],size+1,MPI_DOUBLE,
              mynode,MPI_COMM_WORLD);
    for(j=0;j<size+1;j++)
  tmp[j] = A_local[cnt][j];
    cnt++;
  }
  else{
    MPI_Bcast(tmp,size+1,MPI_DOUBLE,i%totalnodes,
              MPI_COMM_WORLD);
  }
  for(j=cnt;j<numrows;j++){
    scaling = A_local[j][i]/tmp[i];
    for(k=i;k<size+1;k++)
      A_local[j][k] = A_local[j][k] - scaling*tmp[k];
  }
}
```

Remark 1: We use the integer variable *cnt* to keep track of how many rows *on each processor* have been reduced. Recall that each processor has its own copy of the variable *cnt*, and hence on each processor it can be used to keep track of what is the active row.

Remark 2: We have chosen the *cyclic distribution* discussed earlier. A schematic of the communication and computation pattern for a four processor run executed on a *size* = 12 system is shown in figure 9.9.

Part 4 - Preparation for back substitution

```
/* PART 4 */
```

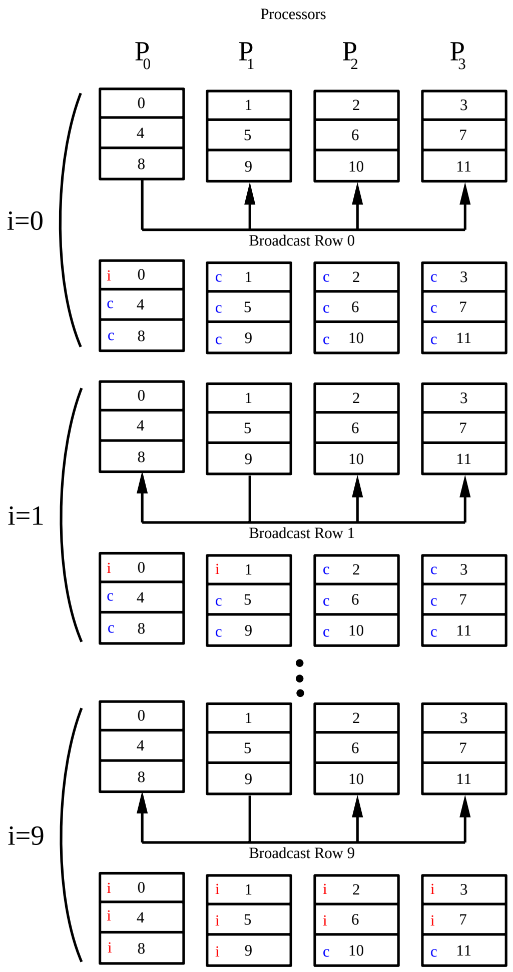

Figure 9.9: Schematic of the communication and computation pattern for a four processor run executed on a $size = 12$ system. Iterations $i = 0$, $i = 1$, and $i = 9$ are shown. The letter **i** denotes an idle row, while a c denotes a computing row.

```
/* On each processor, initialize the value of x as equal to
   the modified (by Gaussian elimination) right-hand-side if
   that information is on the processor, otherwise initialize
   to zero.
*/

cnt = 0;
for(i=0;i<size;i++){
  if(i==myrows[cnt]){
    x[i] = A_local[cnt][size];
    cnt++;
  }
  else
    x[i] = 0;
}
```

Remark: To accomplish the back substitution, we first initialize the solution by setting the array entry $x[i]$ equal to the last column of the augmented matrix (which contains the modified right-hand-side) for those rows for which a processor is responsible, and equal to zero for all rows for which a particular processor is not responsible.

Part 5 - Back substitution

```
/* PART 5 */
/* Backsolve to find the solution x */
cnt = numrows-1;
for(i=size-1;i>0;i--){
  if(cnt>=0){
    if(i == myrows[cnt]){
      x[i] = x[i]/A_local[cnt][i];
      MPI_Bcast(x+i,1,MPI_DOUBLE,mynode,MPI_COMM_WORLD);
      cnt--;
    }
    else
      MPI_Bcast(x+i,1,MPI_DOUBLE,i%totalnodes,MPI_COMM_WORLD);
  }
  else
    MPI_Bcast(x+i,1,MPI_DOUBLE,i%totalnodes,MPI_COMM_WORLD);
    for(j=0;j<=cnt;j++)
      x[myrows[j]] = x[myrows[j]] - A_local[j][i]*x[i];
  }

  if(mynode==0){
    x[0] = x[0]/A_local[cnt][0];
    MPI_Bcast(x,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
  }
  else
    MPI_Bcast(x,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

Remark 1: Observe that the variable *cnt* is initialized to $(numrows - 1)$ and is decremented in this loop. Recall that we want to traverse back up the rows (hence the name back substitution). As a processor computes the solution for a row for which it is responsible, it then broadcasts the result to all the other processors.

Remark 2: Why do we check for $cnt \geq 0$? If you examine this loop carefully, you will notice that at some point the variable *cnt* is equal to -1. This occurs when all the solution components on a processor have been computed. We must verify that $cnt \geq 0$ prior to attempting to access myrows[cnt], otherwise we are performing an illegal memory access, and hence the process may fail. We know, however, that once a processor's value of *cnt* is equal to -1 it merely needs to obtain the updated values from other processors, and hence we can immediately proceed to an *MPI_Bcast* call to obtain the solution from another processor.
Remark 3: Observe in the *MPI_Bcast* arguments that we use pointer arithmetic for updating the address which is passed to the function. Recall that x is a pointer (and hence has as its value an address). The expression $(x + i)$ is equivalent to the expression $\&x[i]$, which can be read as "the address of the array element x[i]."

Part 6 - Program finalization

```
  /* PART 6 */

  if(mynode==0){
    for(i=0;i<size;i++)
      cout << x[i] << endl;
  }
  delete[] tmp;
  delete[] myrows;
  for(i=0;i<numrows;i++)
    delete[] A_local[i];
  delete[] A_local;

  MPI_Finalize();

}


void ChebyVandermonde(int npts, double *A, int row){
  int i,j;
  double * x = new double[npts];

  ChebyshevPoints(npts,x);

  for(j=0;j<npts;j++)
    A[j] = pow(x[row],j);

  delete[] x;
}
```

Remark: We conclude the program by printing the solution from the first processor ($mynode = 0$). If we were not to put the *if* statement there, all processors would print the solution.

9.1.5 Gaussian Elimination and Sparse Systems

Many linear systems that arise from discretization of partial differential equations, as we have seen in chapters 6 and 7, are sparse and more specifically bandeded. Assuming that we deal with symmetrically banded systems of (semi)-bandwidth m, we can modify accordingly the (ijk) loops of table 9.1

to account for this sparsity. For example, the *kij* loop is modified as follows:

```
for $k = 1, n-1$
    for $i = k+1$, min$(k+m, n)$
        $\ell_{ik} = a_{ik}/a_{kk}$
        for $j = k+1$, min$(k+m, n)$
            $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
        endfor
    endfor
endfor
```

The computational complexity of this algorithm is significantly less on a serial computer than on a parallel computer. The operation count for LU decomposition is about $\mathcal{O}(nm^2)$ for an $n \times n$ matrix with bandwidth m and for the backsolve is $\mathcal{O}(nm)$. However, on a parallel computer a straightforward implementation of the above algorithm would result in large inefficiencies. For example, the row- (column-) blocked interleaved scheme discussed earlier becomes very inefficient when $m < P$, where P is the total number of processors as only m processors are effectively used. Clearly, the case with $m = 1$, i.e., the tridiagonal system, is the most difficult case and needs to be handled differently; we study this case next.

9.1.6 Parallel Cyclic Reduction for Tridiagonal Systems

Several algorithms have been developed over the years for the parallel solution of tridiagonal linear systems including *recursive doubling* [82], *cyclic reduction* [13], *domain decomposition* [90], and their many variants. Here we present the cyclic reduction method, which has been one of the most successful approaches.

The main idea of cyclic reduction is to group the unknowns in *even* and *odd-numbered* entries, just like in black/red Gauss-Seidel (see section 7.2.4), and successively eliminate the odd-numbered entries. Most of the operations in this process can be done in parallel. We present here some details by considering a specific small system in order to illustrate how we manipulate the equations.

Let us consider the tridiagonal system

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = F_i, \qquad i = 1, \ldots, n \tag{9.8}$$

where a_i, b_i, c_i, and F_i are given, and we also assume that $n = 2^p - 1$. If $n \neq 2^p - 1$ then we add additional trivial equations of the form $x_i = 0$, $i = n+1, \ldots, 2^p - 1$.

The key idea is then to combine linearly the equations in order to eliminate the odd-numbered unknowns

$$x_1, x_3, x_5, \dots, x_n$$

in the first stage. We then re-order (i.e., re-number) the unknowns and repeat this process until we arrive at a single equation with one unknown. Upon solution of that equation, we march backwards to obtain the rest of the unknowns. To do this we combine the equations in triplets.

Next, we demonstrate this for the case of $n = 7 = 2^3 - 1$ unknowns for which we have three triplets. We start by forming the first triplet from the first three equations. To this end, we multiply by the parameters α_2, β_2, γ_2 to get

$$\begin{aligned}
\alpha_2 b_1 x_1 \quad &+ \quad \alpha_2 c_1 x_2 = \alpha_2 F_1 \\
\beta_2 a_2 x_1 \quad &+ \quad \beta_2 b_2 x_2 + \beta_2 c_2 x_3 = \beta_2 F_2 \\
& \qquad \gamma_2 a_3 x_2 + \gamma_2 b_3 x_3 + \gamma_2 c_3 x_4 = \gamma_2 F_3.
\end{aligned}$$

In order to eliminate x_1 and x_3, we add the equations and choose:

$$\begin{aligned}
&\beta_2 = 1 \\
\alpha_2 b_1 \quad + \quad &\beta_2 a_2 = 0 \\
\beta_2 c_2 \quad + \quad &\gamma_2 b_3 = 0.
\end{aligned}$$

The resulted equation (sum of the above three equations) is

$$\underbrace{(\alpha_2 c_1 + \beta_2 b_2 + \gamma_2 a_3)}_{\hat{b}_2} x_2 + \underbrace{\gamma_2 c_3}_{\hat{c}_2} x_4 = \underbrace{\alpha_2 F_1 + \beta_2 F_2 + \gamma_2 F_3}_{\hat{F}_2}.$$

Similarly, combining again equations, i.e., the third, fourth and fifth equations obtained from equation (9.8) we form the second triplet from which we obtain

$$\underbrace{\alpha_4 a_3}_{\hat{a}_4} x_2 + \underbrace{(\alpha_4 c_3 + \beta_4 b_4 + \gamma_4 a_5)}_{\hat{b}_4} x_4 + \underbrace{\gamma_4 c_5}_{\hat{c}_4} x_6 = \underbrace{a_4 F_3 + \beta_4 F_4 + \gamma_4 F_5}_{\hat{F}_4},$$

and α_4, β_4 and γ_4 are determined from

$$\begin{aligned}
&\beta_4 = 1 \\
\alpha_4 b_3 \quad + \quad &\beta_4 a_4 = 0 \\
\beta_4 c_4 \quad + \quad &\gamma_4 b_5 = 0.
\end{aligned}$$

Finally, for the third triplet we obtain, as before, the only surviving equation

$$\hat{a}_6 x_4 + \hat{b}_6 x_6 = \hat{F}_6,$$

where the parameters α_6, β_6, γ_6 involved in the definition of $\hat{a}_6$, $\hat{b}_6$ and $\hat{F}_6$ are determined by solving

$$\begin{aligned} \alpha_6 b_5 + \beta_6 a_6 &= 0 \\ \beta_6 c_6 + \gamma_b b_7 &= 0. \end{aligned}$$

We see that the three resulted equations also form a tridiagonal system, i.e.,

$$\hat{b}_2 x_2 + \hat{c}_2 x_4 = \hat{F}_2 \tag{9.9}$$

$$\hat{a}_4 x_2 + \hat{b}_4 x_4 + \hat{c}_4 x_6 = \hat{F}_4 \tag{9.10}$$

$$\hat{a}_6 x_4 + \hat{b}_6 x_6 = \hat{F}_6. \tag{9.11}$$

We can repeat the same elimination process as before, i.e., first multiply by $\alpha_4^{'}$, $\beta_4^{'}$ and $\gamma_4^{'}$ respectively the above equations and choose

$$\begin{aligned} \alpha_4^{'} \hat{b}_2 + \beta_4^{'} \hat{a}_4 &= 0 \\ \beta_4^{'} \hat{c}_4 + \gamma_4^{'} \hat{b}_6 &= 0 \end{aligned}$$

that leads to only one equation

$$\alpha_4^* x_4 = F_4^*.$$

Using back substitution, after we obtain x_4 from above, we can compute x_2 from the reduced equation (9.9) and x_6 from equation (9.11). Finally, we use the original equations to obtain x_1, x_3, x_5 and x_7.

In summary, we perform the following steps:

1. Compute

$$(\alpha_2, \beta_2, \gamma_2)$$
$$(\alpha_4, \beta_4, \gamma_4)$$
$$(\alpha_6, \beta_6, \gamma_6)$$

2. Compute

$$(\hat{b}_2, \hat{c}_2, \hat{F}_2)$$
$$(\hat{a}_4, \hat{b}_4, \hat{c}_4, \hat{F}_4)$$
$$(\hat{a}_6, \hat{b}_6, \hat{F}_6)$$

3. Compute
$$(\alpha_4^{'}, \beta_4^{'}, \gamma_4^{'})$$
$$(a_4^*, F_4^*)$$

4. Solve for x_4, x_2, x_6, x_1, x_3, x_5 and x_7.

Remark: There exist non-singular matrices for which the above method will terminate early with the solution. The number of levels used above is the worst-case scenario. For some matrices, the depth of reduction can be truncated due to the solution being obtained for one of the variables. Immediately back-substitution can begin.

The operation count is $\mathcal{O}(13n)$ multiplications compared to $\mathcal{O}(5n)$ for the standard LU decomposition. However, this is the serial work but many of the above computations can be done in parallel. For example, let us assume that we have
$$P = \frac{n-1}{2}$$
processors and we store each triplet on one processor. The work for eliminating the odd-numbered unknowns can be done in parallel, and at the end of this stage each processor holds one reduced equation. To proceed, the processors need to exchange data. This can be done by nearest neighbor communications, e.g., P_2 will receive data from P_1 and P_3, P_4 will receive data from P_3 and P_5, and so on. This implies that about half of the processors (e.g., all the odd-numbered $P_1, P_3 \ldots$) will remain idle. After $(p-1)$ reduction stages (where $n = 2^p - 1$) only one processor will solve the final equation. However, the rest of the processors are not quite retired yet, as in the back substitution they will all be called on duty again!

Software
Suite *Putting it into Practice*

We now present implementations of cyclic reduction for tridiagonal systems. Due to the complex nature of the indexing involved, we will present both the serial and then the parallel version. As pointed out earlier, the serial version of this algorithm is rarely used because it is actually more expensive than standard LU; we present it however because it is easier to understand the complex indexing in the serial setting without the additional complexity of the parallelization.

As a model problem, we solve for the system given in figure 9.10. Notice that the matrix **A** is of the form found in discretizations of the Laplace's equation with second-order finite difference schemes.

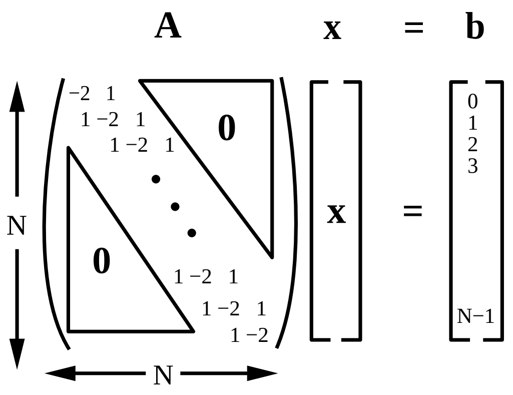

Figure 9.10: Tridiagonal system used as a model problem for cyclic reduction.

To better explain the code, we have broken the entire program into multiple parts. The serial code is broken down into three parts as follows:

1. Part 1 - Memory allocation and generation of the matrix **A**.
2. Part 2 - Cyclic reduction stages.
3. Part 3 - Cyclic reduction back substitution to recover the solution.

For each part, we will first present the code and then present a collection of remarks elucidating the salient points within each part.

$\boxed{\text{Serial}}$

```
#include <iostream.h>
#include <iomanip.h>
#include "SCmathlib.h"


const int size = 15;

void main(){
```

```
int i,j,k;
int index1,index2,offset;
double alpha,gamma;
```

Part 1 - Memory allocation and generation of matrix

```
/* Part 1 */
double * x = new double[size];
for(i=0;i<size;i++)
  x[i] = 0.0;
double * F = new double[size];
double ** A = new double*[size];


for(i=0;i<size;i++){
  A[i] = new double[size];
  for(j=0;j<size;j++)
    A[i][j] = 0.;
  F[i] = (double)i;
}

A[0][0] = -2.0; A[0][1] = 1.0;
A[size-1][size-2] = 1.0; A[size-1][size-1] = -2.0;

for(i=1;i<size-1;i++){
  A[i][i] = -2.0;
  A[i][i-1] = 1.0;
  A[i][i+1] = 1.0;
}
```

Remark 1: Observe that in this program we use more memory than necessary. Because we know that we are operating on a tridiagonal system, the memory required is greatly reduced from that which would be needed in **A** were full. Here we allocate memory as if **A** is a full matrix; we leave it as an exercise to modify this program to use only the necessary amount of memory.

Part 2 - Cyclic reduction

```
/* Part 2 */
for(i=0;i<log2(size+1)-1;i++){
    for(j=pow(2,i+1)-1;j<size;j=j+pow(2,i+1)){
      offset = pow(2,i);
      index1 = j - offset;
      index2 = j + offset;

      alpha = A[j][index1]/A[index1][index1];
      gamma = A[j][index2]/A[index2][index2];

      for(k=0;k<size;k++){
        A[j][k] -= (alpha*A[index1][k] + gamma*A[index2][k]);
      }
      F[j] -= (alpha*F[index1] + gamma*F[index2]);
    }
  }
```

Remark 2: In the above code section, the first *for* loop (over *i*) iterates through the levels of reduction which occur, while the second *for* loop (over *j*) indexes which rows at each level are to be acted upon. Notice that both loops and the variables *index1* and *index2* are based on powers of two.

Part 3 - Back substitution

```
  /* Part 3 */
  int index = (size-1)/2;
  x[index] = F[index]/A[index][index];

  for(i=log2(size+1)-2;i>=0;i--){
    for(j=pow(2,i+1)-1;j<size;j=j+pow(2,i+1)){
      offset = pow(2,i);
      index1 = j - offset;
      index2 = j + offset;

      x[index1] = F[index1];
      x[index2] = F[index2];
      for(k=0;k<size;k++){
        if(k!= index1)
          x[index1] -= A[index1][k]*x[k];
        if(k!= index2)
          x[index2] -= A[index2][k]*x[k];
      }
      x[index1] = x[index1]/A[index1][index1];
      x[index2] = x[index2]/A[index2][index2];
    }
  }

  for(i=0;i<size;i++){
    cout << x[i] << endl;
  }

  delete[] x;
  delete[] F;
  for(i=0;i<size;i++)
    delete[] A[i];
  delete[] A;
}
```

Remark 3: Once the full reduction has occurred, we must traverse back up the reduction tree. Note that the two *for* loops accomplish this traversal.

Parallel

We now present the parallel version of the serial code presented above. We assume that given P processors, we will accomplish cyclic reduction on a matrix of size $2^{\log_2(P+1)+1} - 1$. This amounts to associating three rows per processor during the first stage. The parallel code is broken down into four parts as follows:

1. Part 1 - MPI initialization and both memory allocation and generation of the matrix **A**.
2. Part 2 - Parallel cyclic reduction stages.
3. Part 3 - Parallel cyclic reduction back substitution to distribute necessary information.
4. Part 4 - Solution for the odd rows for which each process is responsible.

For each part, we will first present the code and then present a collection of remarks elucidating the salient points within each part.

Part 1 - MPI initialization

```
#include <iostream.h>
#include <iomanip.h>
#include "SCmathlib.h"
#include<mpi.h>


int main(int argc, char *argv[]){
  int i,j,k,size,index;
  int index1,index2;
  int mynode, totalnodes;
  double alpha,gamma;
  const int numrows = 5;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  size = (int) pow(2,log2(totalnodes+1)+1)-1;

  double ** A = new double*[numrows];
  for(i=0;i<numrows;i++){
    A[i] = new double[size+1];
    for(j=0;j<size+1;j++)
      A[i][j] = 0.0;
  }

  if(mynode==0){
    A[0][0] = -2.0; A[0][1] = 1.0;
    A[1][0] = 1.0; A[1][1] = -2.0; A[1][2] = 1.0;
    A[2][1] = 1.0; A[2][2] = -2.0; A[2][3] = 1.0;
  }
  else if(mynode==(totalnodes-1)){
    index = 2*mynode;
    A[0][index-1] = 1.0; A[0][index] = -2.0;
    A[0][index+1] = 1.0;
    index = 2*mynode+1;
    A[1][index-1] = 1.0; A[1][index] = -2.0;
    A[1][index+1] = 1.0;
    A[2][size-2] = 1.0; A[2][size-1] = -2.0;
  }
  else{
    for(i=0;i<3;i++){
      index = i + 2*mynode;
      A[i][index-1] = 1.0;
      A[i][index]   = -2.0;
      A[i][index+1] = 1.0;
    }
  }

  for(i=0;i<3;i++)
    A[i][size] = 2*mynode+i;


  int numactivep = totalnodes;
  int * activep = new int[totalnodes];
  for(j=0;j<numactivep;j++)
    activep[j] = j;


  for(j=0;j<size+1;j++){
    A[3][j] = A[0][j];
    A[4][j] = A[2][j];
  }
```

Remark 1: Just as in the parallel Gaussian elimination code, we augment the matrix **A** with the right-hand-side (appending **A** with an extra column). This helps to minimize the communication by allowing us to communicate both the row and right-hand-side information simultaneously.

Part 2 - Cyclic reduction

```
 /* Part 2 */

  for(i=0;i<log2(size+1)-1;i++){
    for(j=0;j<numactivep;j++){
      if(mynode==activep[j]){
        index1 = 2*mynode + 1 - pow(2,i);
        index2 = 2*mynode + 1 + pow(2,i);
        alpha = A[1][index1]/A[3][index1];
        gamma = A[1][index2]/A[4][index2];

        for(k=0;k<size+1;k++)
          A[1][k] -= (alpha*A[3][k] + gamma*A[4][k]);


        if(numactivep>1){
          if(j==0){
    MPI_Send(A[1],size+1,MPI_DOUBLE,activep[1],0,
                     MPI_COMM_WORLD);
  }
  else if(j==numactivep-1){
    MPI_Send(A[1],size+1,MPI_DOUBLE,activep[numactivep-2],
                     1,MPI_COMM_WORLD);
  }
  else if(j%2==0){
    MPI_Send(A[1],size+1,MPI_DOUBLE,activep[j-1],
                     1,MPI_COMM_WORLD);
    MPI_Send(A[1],size+1,MPI_DOUBLE,activep[j+1],
                     0,MPI_COMM_WORLD);
  }
  else{
    MPI_Recv(A[3],size+1,MPI_DOUBLE,activep[j-1],0,
                     MPI_COMM_WORLD,&status);
    MPI_Recv(A[4],size+1,MPI_DOUBLE,activep[j+1],1,
                     MPI_COMM_WORLD,&status);
  }
        }
      }
    }

    numactivep = 0;
    for(j=activep[1];j<totalnodes;j=j+pow(2,i+1)){
      activep[numactivep++]=j;
    }
  }
```

Remark 2: The communication is accomplished through a series of *MPI_Send* and *MPI_Recv* calls. Each processor is communicating (either sending or receiving) from at most two other processors. To keep track of whom is to be sending/receiving, we maintain an active processor list within the integer array *activep*. A communication schematic for cyclic reduction using seven processors is given in figure 9.11.

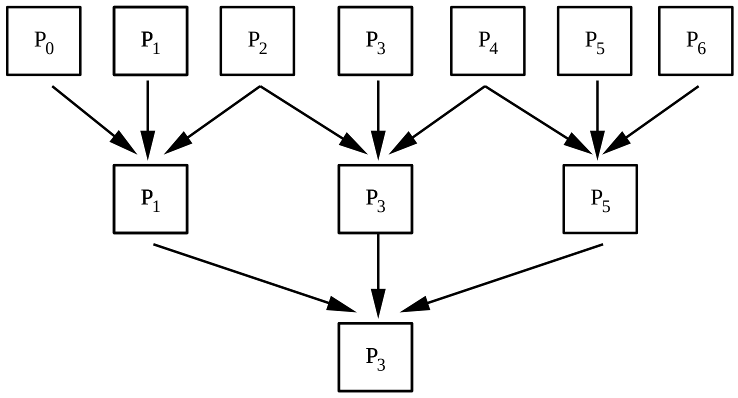

Figure 9.11: Cyclic reduction communication pattern for seven processor case.

Part 3 - Back substitution

```
/* Part 3 */

  double * x = new double[totalnodes];
  for(j=0;j<totalnodes;j++)
    x[j] = 0.0;

  if(mynode==activep[0]){
    x[mynode] = A[1][size]/A[1][(size-1)/2];
  }


  double tmp;
  for(i=log2(size+1)-3;i>=0;i--){
    tmp = x[mynode];
    MPI_Allgather(&tmp,1,MPI_DOUBLE,x,1,MPI_DOUBLE,
                  MPI_COMM_WORLD);
    numactivep = 0;
    for(j=activep[0]-pow(2,i);j<totalnodes;j=j+pow(2,i+1)){
      activep[numactivep++]=j;
    }

    for(j=0;j<numactivep;j++){
      if(mynode == activep[j]){
        x[mynode] = A[1][size];
        for(k=0;k<totalnodes;k++){
          if(k!=mynode)
            x[mynode] -= A[1][2*k+1]*x[k];
        }
        x[mynode] = x[mynode]/A[1][2*mynode+1];
      }
    }
  }

  tmp = x[mynode];
  MPI_Allgather(&tmp,1,MPI_DOUBLE,x,1,MPI_DOUBLE,
                MPI_COMM_WORLD);
```

Remark 3: A schematic for the backward solve communication is given in figure 9.12. Notice that is varies slightly from that of the forward part of the reduction. After a processor has found the solution for its row and has communicated that information to the appropriate processors, it is no longer active. This can be observed in figure 9.12 - observe that processor P_3 no longer has things to compute in the second and third levels.

Remark 4: We use the *MPI_Allgather* command so that at any given level all the processors have the available solution up to that point. This all inclusive communication could be replaced by *MPI_Send*/*MPI_Recv* pairs where only those processors requiring particular information would be updated.

Part 4 - Solving for odd rows

```
 /* Part 4 */
  for(k=0;k<totalnodes;k++){
    A[0][size] -= A[0][2*k+1]*x[k];
    A[2][size] -= A[2][2*k+1]*x[k];
  }
  A[0][size] = A[0][size]/A[0][2*mynode];
```

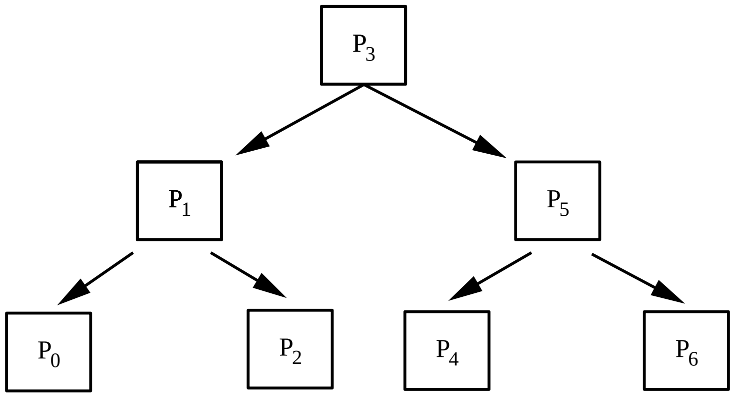

Figure 9.12: A schematic for the backward solve communication when seven processors are used.

```
  A[1][size] = x[mynode];
  A[2][size] = A[2][size]/A[2][2*mynode+2];


  delete[] activep;
  for(i=0;i<numrows;i++)
    delete[] A[i];
  delete[] A;
  delete[] x;


  MPI_Finalize();
}
```

Remark 5: The program concludes with each processor computing the solution for the odd rows for which it was responsible. If the total solution vector were needed on all processors, *MPI_Allgather* could be used to collect the solution on each processor. Note that some additional logic would be necessary to properly take into account the overlap in row distribution (i.e., both processor zero and processor one solve for the solution of matrix row number three) as shown in figure 9.13.

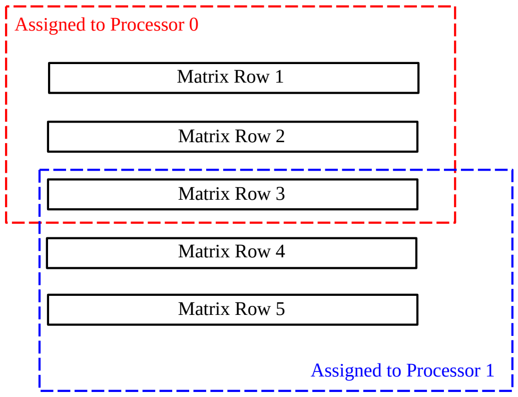

Figure 9.13: Overlap in row solutions for two processor case.

9.2 Cholesky Factorization

A special case of the LU decomposition for a *symmetric positive-definite* matrix $\mathbf{A}$ can be obtained in the form

$$\mathbf{A} = \mathbf{L}\,\mathbf{L}^T.$$

Here, the matrix $\mathbf{U}$ is equal to the transpose of $\mathbf{L}$ due to symmetry. There are many possibilities for $\mathbf{L}$ but this factorization is unique if we require that all diagonal elements of $\mathbf{L}$ be positive. We note that in this case $\mathbf{L}$ is different than the matrix we obtain in the LU decomposition, where all *diagonal elements* are equal to one.

Instead of following the standard LU decomposition, we can obtain explicitly the elements ℓ_{ij} of $\mathbf{L}$ by setting:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & & & \\ \vdots & & & \\ \vdots & & & \\ a_{n1} & \dots & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} \ell_{11} & & & \text{O} & \\ \ell_{12} & \ell_{22} & & & \\ \vdots & \vdots & \ddots & & \\ \vdots & \vdots & & \ddots & \\ \ell_{n1} & \ell_{n2} & \dots & & \ell_{nn} \end{bmatrix} \begin{bmatrix} \ell_{11} & \ell_{21} & \dots & & \ell_{n1} \\ & \ell_{22} & & & \ell_{n2} \\ & & \ddots & & \vdots \\ \text{O} & & & \ddots & \\ & & & & \ell_{nn} \end{bmatrix}$$

Next, we equate elements on both sides to obtain

$$a_{i1} = \ell_{i1}\ell_{11} \quad \text{and} \quad a_{11}^2 = \ell_{11}^2$$

for the entire first column ($i = 1, \dots, n$), and similarly for the other columns. The following pseudo-code summarizes the Cholesky algorithm:

```
for j = 1, n
    $\ell_{ij} = \sqrt{a_{ij} - \sum_{k=1}^{i-1} \ell_{jk}^2}$
    for i = j + 1, n
        $\ell_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} \ell_{ik}\ell_{jk}}{\ell_{jj}}$
    endfor
endfor
```

Remark 1: The Cholesky factorization algorithm is stable and thus it does not require pivoting [25].

Remark 2: The Cholesky algorithm requires about half the memory and about half the operations of the LU decomposition.

Remark 3: The positive-definite property is important in obtaining the ℓ_{ij} without partial pivoting. In fact, partial pivoting can destroy the symmetry of a matrix $\mathbf{A}$.

Remark 4: In some cases an *incomplete* or *approximate* Cholesky decomposition is required, e.g. as a preconditioner in accelerating the convergence of iterative solvers, see section 9.4.2. This is achieved by simply filling in with zeros the entries of $\mathbf{L}$, which have corresponding zero entries in the original (presumably sparse) matrix $\mathbf{A}$.

9.3 QR Factorization and Householder Transformation

The LU factorization is not the only way of factorizing a matrix $\mathbf{A}$. The Householder transformation we present here is the basis of an efficient factorization of a general matrix

$$\mathbf{A} = \mathbf{QR},$$

where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix. We have already studied in section 2.2.9 how to achieve such a QR decomposition by orthogonalizing vectors via the Gram-Schmidt procedure, but the procedure we present here is always stable and much more efficient.

We start by considering the following important operation in scientific computing:

- *How to take a full vector and produce a special vector with only one of its entries non-zero.*

This is accomplished efficiently in terms of the (*orthogonal*) Householder matrix, which is defined by

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{w}\mathbf{w}^T}{\mathbf{w}^T\mathbf{w}} \quad \forall \mathbf{w} \neq 0.$$

Of particular interest is the vector

$$\alpha \mathbf{e}_1 = (\alpha, 0, 0, \ldots 0)^T$$

which can be created from an arbitrary vector $\mathbf{x}$ if the vector $\mathbf{w}$ is computed appropriately. To this end, we set $\mathbf{w}$ such that:

$$\mathbf{H}\,\mathbf{x} = \left(\mathbf{I} - 2\frac{\mathbf{w}\mathbf{w}^T}{\mathbf{w}^T\mathbf{w}}\right)\mathbf{x} = \begin{bmatrix} \alpha \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \alpha \mathbf{e}_1 \,.$$

The solution to this problem is simple, and it is given by

$$\mathbf{w} = \mathbf{x} + sign(x_1) \cdot ||\mathbf{x}||_2 \mathbf{e}_1,$$

where x_1 is the first entry of the vector $\mathbf{x}$. This transformation from $\mathbf{H}\,\mathbf{x} \rightarrow (\alpha, 0, 0, \ldots, 0)^T$ is called *Householder transformation.*

First, we summarize the algorithm that describes the Householder transformation in the following pseudo-code:

```
$x_m = \max\{|x_1|, |x_2|, \ldots, |x_n|\}$
for $k = 1, n$
    $w_k = x_k / x_m$
endfor
$\alpha = sign(w_1)[w_1^2 + w_2^2 + \ldots + w_n^2]^{1/2}$
$w_1 = w_1 + \alpha$
$\alpha = -\alpha x_m$
```

Then, the desired vector is $(\alpha, 0, 0, \ldots 0)^T$. The number of operations is proportional to $\mathcal{O}(n)$, that is it takes *linear work* only to accomplish this important operation.

Example: Consider the vector

$$\mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix},$$

then $x_m = 3$, and the intermediate values of the components of $\mathbf{w}$ are:

$$w_1 = \frac{1}{3}; \quad w_2 = \frac{2}{3}; \quad w_3 = -1.$$

Also, the intermediate value of α is:

$$\alpha = +\sqrt{\frac{1}{3^2} + \frac{2^2}{3^2} + (-1)^2} = 1.2472.$$

The updated values are then

$$w_1 = \frac{1}{3} + 1.2472 = 1.5805; \quad w_2 = \frac{2}{3}; \quad w_3 = -1$$

and

$$\alpha = -1.2472 \cdot 3 = -3.7416$$

while the desired vector is

$$\mathbf{Hx} = \begin{pmatrix} -3.7416 \\ 0 \\ 0 \end{pmatrix}.$$

Remark 1: The matrix-vector product with a Householder matrix

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{w}\mathbf{w}^T}{\mathbf{w}^T\mathbf{w}}$$

is only an $\mathcal{O}(n)$ operation compared to the $\mathcal{O}(n^2)$ for a general matrix-vector product. This can be achieved from the relation

$$\mathbf{H}\,\mathbf{x} = \mathbf{x} - \beta\mathbf{w}(\mathbf{w}^T\mathbf{x}),$$

where $\beta^{-1} = \mathbf{w}^T\mathbf{w}/2$ is a scalar. We note that the right-hand-side is computed within one loop

$$x_i = x_i - \beta \cdot \gamma \cdot w_i, \; i = 1., \ldots, n$$

where $\gamma = \mathbf{w}^T\mathbf{x}$ is also scalar.

Clearly, we do not need to construct explicitly $\mathbf{H}$ in this case – that cost would lead to an $\mathcal{O}(n^2)$ operation!

To accomplish the QR factorization of a square general $n \times n$ matrix $\mathbf{A}$, we consider its columns and apply successively the Householder transformation in order to zero out the subdiagonal entries of each column. We do this in $(n-1)$ stages, just like in LU decomposition.

Stage 1: We consider the *first column* of $\mathbf{A}$ and determine a Householder matrix $\mathbf{H}_1$ so that:

$$\mathbf{H}_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

To determine $\mathbf{H}_1$ we simply need to apply the Householder transformation algorithm to obtain $\mathbf{w}_1$ of length n. After the first stage we overwrite $\mathbf{A}$ by $\mathbf{A}_1$ where

$$\mathbf{A}_1 = \begin{pmatrix} \alpha_1 & a_{12}^* & \dots & a_{1n}^* \\ 0 & a_{22}^* & \dots & \dots \\ \vdots & \vdots & & \\ 0 & a_{n2}^* & \dots & a_{nn}^* \end{pmatrix} = \mathbf{H}_1\mathbf{A},$$

which has all new elements (denoted by star) after the first column.

Stage 2: Next we consider the *second column* of the updated matrix

$$\mathbf{A} \equiv \mathbf{A}_1 = \mathbf{H}_1\mathbf{A}$$

and take only the part below the diagonal, to obtain

$$\mathbf{H}_2^* \begin{bmatrix} a_{22}^* \\ a_{32}^* \\ \vdots \\ a_{n2}^* \end{bmatrix} = \begin{bmatrix} \alpha_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

which yields a vector $\mathbf{w}_2$ of length $(n-1)$. This vector defines uniquely the Householder matrix

$$\mathbf{H}_2^* = \mathbf{I} - 2\frac{\mathbf{w}_2\mathbf{w}_2^T}{\mathbf{w}_2^T\mathbf{w}_2}.$$

Unlike $\mathbf{H}_1$, here we first need to "inflate" $\mathbf{H}_2^*$ to $\mathbf{H}_2$ and then overwrite $\mathbf{A}$ by $\mathbf{H}_2\mathbf{A}_1$, where

$$\mathbf{H}_2 = \begin{pmatrix} 1 & \dots & 0 \\ 0 & & \mathbf{H}_2^* \end{pmatrix}.$$

Stage k: In the k-stage of the QR procedure we produce a vector $\mathbf{w}_k$ of length $(n-k+1)$ by solving

$$\mathbf{H}_k^* \begin{bmatrix} a_{kk}^* \\ a_{k+1,k}^* \\ \vdots \\ a_{nk}^* \end{bmatrix} = \begin{bmatrix} \alpha_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Here again, we overwrite $\mathbf{A}$ by

$$\mathbf{A}_k = \mathbf{H}_k \mathbf{A}_{k-1}\,,$$

and subsequently we "inflate" $\mathbf{H}_k^*$ as

$$\mathbf{H}_k = \begin{bmatrix} \mathbf{I}_{k-1} & 0 \\ 0 & \mathbf{H}_k^* \end{bmatrix}.$$

Remark 2: The efficiency of the Householder algorithm is based on the efficient multiplication

$$\mathbf{A}_{k+1} = \mathbf{H}_{k+1} \cdot \mathbf{A}_k\,,$$

which should not be performed explicitly but rather using the $\mathcal{O}(n)$ matrix-vector product algorithm we presented above; that is we compute

$$(\mathbf{I} - \beta \mathbf{w}_{n-k}\mathbf{w}_{n-k}^T)[\mathbf{a}_{kj}]$$

where

$$\beta^{-1} = \mathbf{w}_{n-k}^T \cdot \mathbf{w}_{n-k}/2,$$

and $\mathbf{a}_{kj}$ denotes the columns of $\mathbf{A}_k$ with $j = k+1, \dots, n-k$. The determination of $\mathbf{H}_k^*$ requires $(n-k)$ operations because the unknown vector $\mathbf{w}$ is of $(n-k)$ length.

Throughout the following section, the subscript for the vector $\mathbf{w}$ will denote the current size of the vector; hence $\mathbf{w}_{n-k}$ denotes the vector of size $(n-k)$. In the algorithm to be presented, the notation w_{ij} denotes the i^{th} entry of the vector $\mathbf{w}_j$.

After a total of $(n-1)$ stages we obtain an upper triangular matrix $\mathbf{R}$ with diagonal elements the α_k $(k = 1, \ldots, n)$ and the other entries computed from:

$$r_{ij} = a_{ij} - \gamma w_{ik}$$

where $n \geq i$, $j \geq k$, and

$$\begin{aligned} \gamma &= \beta \sum_{i=k}^{n} w_{ik} a_{ij} \\ \beta^{-1} &= (\mathbf{w}_{n-k+1}^T \cdot \mathbf{w}_{n-k+1})/2. \end{aligned}$$

We note that the above formulas compute all the entries above the diagonal but also the r_{nn} entry.

Then, the matrix $\mathbf{R}$ is:

$$\mathbf{R} = \begin{bmatrix} \alpha_1 & r_{12} & r_{13} & \ldots & \ldots & r_{1n} \\ & \alpha_2 & r_{23} & \ldots & \ldots & r_{2n} \\ & & \alpha_3 & \ldots & \ldots & r_{3n} \\ & & & \ddots & & \\ & \mathrm{O} & & & \alpha_{n-1} & \\ & & & & & r_{nn} \end{bmatrix}.$$

This matrix can be constructed by forming an equivalent Householder matrix $\mathbf{H}_k$ of size n at each stage from the $\mathbf{H}_k^*$ which has order $(n-k+1)$. To this end, we simply set

$$\mathbf{H}_k = \begin{bmatrix} \mathbf{I}_{k-1} & O \\ O & \mathbf{H}_k^* \end{bmatrix},$$

and we also compute:

$$\mathbf{A}_k = \mathbf{H}_k \mathbf{A}_{k-1}$$

with

$$\mathbf{A}_1 \equiv \mathbf{A}$$

The upper triangular matrix is then

$$\mathbf{R} = \mathbf{A}_{n-1} = \mathbf{H}_{n-1}\mathbf{A}_{n-2} = \ldots = \mathbf{H}_{n-1}\mathbf{H}_{n-2}\ldots\mathbf{H}_1\mathbf{A}\,.$$

We can invert the above equation if we set

$$\mathbf{Q}^T = \mathbf{H}_{n-1}\mathbf{H}_{n-2}\ldots\mathbf{H}_1\,,$$

then

$$\mathbf{Q}^{-1} = \mathbf{Q}^T$$

because $\mathbf{H}_k$ are all orthogonal. Thus, $\mathbf{Q} \cdot \mathbf{R} = \mathbf{Q}\mathbf{Q}^T\mathbf{A} \Rightarrow \mathbf{A} = \mathbf{QR}$. We can now compute the orthogonal matrix $\mathbf{Q}$, i.e., $\mathbf{Q} = \mathbf{H}_1^T\mathbf{H}_1^T \ldots \mathbf{H}_{n-1}^T$.

We have thus obtained a QR decomposition of an $n \times n$ matrix $\mathbf{A}$, similar to the Gram-Schmidt procedure but at *reduced cost*, since we are operating with shorter and shorter vectors in each stage. We first provide the pseudo-code for the Householder QR decomposition and then we will compute the exact operation count. The pseudo-code below returns only the value of $\mathbf{R}$. The matrix $\mathbf{Q}$ can be formed from the above equation using the matrix $\mathbf{H}_i$ formed by its vector $\mathbf{w}$ from the Householder transformation.

Householder Algorithm

```
Begin Loop: $k = 1, \ldots, n-1$ (number of stages)
  •Solve $\mathbf{H}_k^* \begin{bmatrix} a_{kk}^* \\ a_{k+1,k}^* \\ \vdots \\ a_{nk}^* \end{bmatrix} = \begin{bmatrix} \alpha_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ (Obtain $\mathbf{w}_{n-k+1}$)
      •$r_{kk} = \alpha_k$
      •Compute $\beta^{-1} = \mathbf{w}_{n-k+1}^T \cdot \mathbf{w}_{n-k+1}/2$
      Zero $a_{ik}$: $i = k+1, \ldots, n$
      Begin Loop: $j = k+1, \ldots, n$
        $\gamma_j = 0$
        Begin Loop: $q = k, \ldots n$
          $\gamma_j = \gamma_j + \beta w_{qk} a_{qj}$
        End Loop
        Begin Loop: $i = k, \ldots, n$
          $r_{ij} = a_{ij} - \gamma_j w_{ik}$
          $a_{ij} = r_{ij}$
        End Loop
      End Loop
End Loop
```

Example: Let us consider the 3×3 Hilbert matrix

$$\mathbf{A} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

and apply the Householder QR algorithm.

• In the *first stage* ($k = 1$) we solve

$$\mathbf{H}_1 \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \mathbf{w}_1 = \begin{bmatrix} 2.1666 \\ 0.5 \\ 0.3333 \end{bmatrix}$$

and also $r_{11} = \alpha_1 = -1.1666$ and

$$\beta = \frac{2}{\mathbf{w}_1^T \mathbf{w}_1} = 0.3956\,.$$

Then, for $j = 2$, we calculate:

$$\gamma_2 = \beta[w_{11}a_{12} + w_{21}a_{22} + w_{31}a_{32}] = 0.5274$$

and thus

$$\begin{aligned} a_{12} := r_{12} &= a_{12} - \gamma_2 w_{11} = -0.6429 \\ a_{22} := r_{22} &= a_{22} - \gamma_2 w_{21} = 0.0696 \\ a_{32} := r_{32} &= a_{32} - \gamma_2 w_{31} = 0.0795\,. \end{aligned}$$

In the next iteration, $j = 3$, we calculate similarly:

$$\gamma_3 = \beta[w_1 a_{13} + w_2 a_{23} + w_3 a_{33}] = 0.3615$$

and thus

$$\begin{aligned} a_{13} := r_{13} &= a_{13} - \gamma_3 w_{11} = -0.4500 \\ a_{23} := r_{23} &= a_{23} - \gamma_3 w_{21} = 0.0692 \\ a_{33} := r_{33} &= a_{33} - \gamma_3 w_{31} = 0.0795\,. \end{aligned}$$

Also, we have that $a_{21} = a_{31} = 0$.

- In the *second stage* ($k = 2$) we solve

$$\mathbf{H}_2^* \begin{bmatrix} 0.0696 \\ 0.0795 \end{bmatrix} = \begin{bmatrix} \alpha_2 \\ 0 \end{bmatrix} \Rightarrow \mathbf{w}_2 = \begin{bmatrix} 2.3095 \\ 1.0000 \end{bmatrix}.$$

Also, $r_{22} = \alpha_2 = -0.1017$, and

$$\beta = 2/(\mathbf{w}_2^T \cdot \mathbf{w}_2) = 0.315759\,.$$

Then, for $j = 3$, we calculate:

$$\begin{aligned} \gamma_3 &= \beta[w_{22}a_{23} + w_{32}a_{33}] = 0.0756 \\ r_{23} &= a_{23} - \gamma_3 w_{22} = -0.1053 \\ r_{33} &= a_{33} - \gamma_3 w_{32} = 0.0039\,, \end{aligned}$$

where we note that a_{23} and a_{33} are the updated values, which were modified in the first stage.

At the conclusion of this example, we now have the resulting $\mathbf{R}$ matrix of the $\mathbf{QR}$ decomposition, and we also have the Householder transformation vectors $\mathbf{w}$ from which we can form $\mathbf{Q}$.

Computational Cost: The computational complexity of the QR decomposition, as described above, is determined by the cost of computing the matrix $\mathbf{H}_k^*$ which is $\mathcal{O}(n-k)$, and also of computing $\mathbf{A}_k$ from $\mathbf{A}_k = \mathbf{H}_k\mathbf{A}_{k-1}$ which requires $\mathcal{O}((n-k)^2)$ operations. Thus, the combined cost is

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

Calculating more carefully the constant factor and accounting for both additions and multiplications we have

$$\mathcal{O}\left(\frac{4}{3}n^3\right) \text{ QR versus } \mathcal{O}\left(\frac{2}{3}n^3\right) \text{ LU decomposition}\,.$$

Remark 3: After we obtain the QR factorization of $\mathbf{A}$ we can solve a linear system $\mathbf{Ax} = \mathbf{b}$ as follows

$$\mathbf{Ax} = \mathbf{b} \Rightarrow \mathbf{QRx} = \mathbf{b} \Rightarrow \mathbf{Q}^T\mathbf{QRx} = \mathbf{Q}^T\mathbf{b} \Rightarrow \mathbf{Rx} = \mathbf{Q}^T\mathbf{b},$$

which is an upper triangular system and can be solved by back substitution.

Remark 4: It can be shown that the QR decomposition based on the Householder transformation is always stable, see [47].

Remark 5: The QR-Householder decomposition is about *twice* as expensive as the LU decomposition and it also expands the bandwidth of a sparse matrix **A**. In contrast, the LU decomposition preserves the bandwidth but it may be susceptible to numerical instabilities as demonstrated earlier. However, even with partial pivoting the LU decomposition is more efficient than the QR decomposition for large matrices **A**.

Remark 6: (*Givens Rotations*) A third way of obtaining the QR decomposition of a matrix **A** (in addition to Gram-Schmidt and Householder) is to employ the Givens rotation matrix

$$\mathbf{R}(\theta) \equiv \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix},$$

which is orthonormal. By setting

$$\mathbf{R}(\theta) \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} (x^2+y^2)^{1/2} \\ 0 \end{bmatrix}$$

we obtain

$$\cos\theta = \frac{x}{\sqrt{x^2+y^2}}; \quad \sin\theta = \frac{y}{\sqrt{x^2+y^2}}.$$

Based on this idea we construct a general *rotation* matrix

$$\mathbf{R}(\theta; i, j) = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & \cos\theta & \sin\theta & & O \\ & & & & & \\ & & -\sin\theta & \cos\theta & & \\ & O & & & \ddots & \\ & & & & & 1 \end{bmatrix}$$

that can be employed to zero out *one entry* in each iteration cycle instead of a column as in the Householder transformation. However, the Givens rotation is *twice more expensive* than the Householder and *four times* more expensive than the LU. It is not used in solving *square* linear systems but it is used in solutions of *least squares* linear systems and also in eigensolvers.

Remark 7: The stability of the Householder method (also the Givens rotation) is due to the fact that the **Q** matrix is a product of orthogonal matrices, which have condition number equal to one.

9.3.1 Hessenberg and Tridiagonal Reduction

We now consider the transformation of a matrix $\mathbf{A}$ to an upper triangular which also has the first lower diagonal *non-zero*. Specifically, this matrix has the form

$$\mathbf{H_e} \equiv \begin{bmatrix} * & * & * & \dots & * \\ & * & & \dots & * \\ & \ddots \, \ddots & & \vdots & \\ & & & * & * \\ & O & & * & * \end{bmatrix},$$

and it is called *upper Hessenberg matrix.*

The Householder transformation procedure can also be used to obtain this for general matrices. If the matrix is also *symmetric* then the resulted matrix is tridiagonal.

The reduction algorithm involves $(n-2)$ stages of elimination. We want to obtain

$$\mathbf{H_e} = \mathbf{H} \cdot \mathbf{A} \cdot \mathbf{H}^T,$$

where the matrix $\mathbf{H} = \mathbf{H}_{n-2} \cdot \mathbf{H}_{n-1} \dots \mathbf{H}_1$. These are Householder matrices, which are computed from zeroing out sub-columns of $\mathbf{A}$ and its updated versions, i.e.,

$$\mathbf{H}_1 = \begin{bmatrix} \mathbf{I}_1 & 0 \\ 0 & \mathbf{H}_1^* \end{bmatrix},$$

where

$$\mathbf{H}_1^* \begin{bmatrix} a_{21} \\ a_{31} \\ \vdots \\ a_{n1} \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

and

$$\mathbf{A}_1 = \mathbf{H}_1 \mathbf{A} \mathbf{H}_1^T$$

and so on ...

Clearly, $\mathbf{A}$ has a new first column with $a_{11} \neq 0$ and $a_{21} \neq 0$ but all other entries are equal to zero.

Example: We consider again the 3×3 Hilbert matrix

$$\mathbf{A} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

which we transform into an upper Hessenberg matrix in one (3-2) stage. Since $\mathbf{A}$ is also symmetric we expect a tridiagonal matrix $\mathbf{H_e}$.

- In the *first stage* ($k = 1$) we solve

$$\mathbf{H}_1^* \begin{bmatrix} \frac{1}{2} \\ \frac{1}{3} \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix}$$

which gives

$$\mathbf{w}_2 = \begin{bmatrix} 2.2019 \\ 0.6666 \end{bmatrix}$$

and $\alpha = -0.6009$.

We then obtain the new entries of $\mathbf{A}$ using $a_{ij} = a_{ij} - \gamma_i w_j$:

$$\mathbf{H_e} = \mathbf{H}_1 \mathbf{A} \mathbf{H}_1^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.8321 & -0.5547 \\ 0 & -0.5547 & 0.8321 \end{bmatrix}$$

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.8321 & -0.5547 \\ 0 & -0.5547 & 0.8321 \end{bmatrix} = \begin{bmatrix} 1.0 & -0.6009 & 0.0 \\ -0.6009 & 0.5231 & -0.0346 \\ 0.0 & -0.0346 & 0.0103 \end{bmatrix}.$$

Remark 1: The solution of the Hessenberg linear system

$$\mathbf{H_e} \cdot \mathbf{x} = \mathbf{b}$$

can be obtained with Gaussian elimination with partial pivoting, and it is guaranteed to be stable [94]. The computational complexity of this is $\mathcal{O}(n^2)$ including the partial pivoting cost.

Software Suite ⊙ *Putting it into Practice*

We now present a function for computing the upper Hessenberg matrix given a matrix **A**. The function takes as input the matrix **A** and upon completion returns the upper Hessenberg matrix in place of the matrix **A**.

```
void Hessenberg(SCMatrix &A){
  int i,j,k,q;
  double beta,gamma;
  int N = A.Rows();
  SCVector *x = new SCVector(N),
    *w = new SCVector(N);

  for(k=0;k<N-2;k++){
    A.GetColumn(k,*x,k+1);
    A(k+1,k) = HouseholderTrans(*x,*w);
    beta = 2.0/dot(N-k-1,*w,*w);
    for(i=k+2;i<N;i++)
      A(i,k) = (*w)(i-k);
    for(j=k+1;j<N;j++){
      gamma = 0.0;
      for(q=k+1;q<N;q++)
        gamma += beta*(*w)(q-k-1)*A(q,j);
      for(i=k+1;i<N;i++){
        A(i,j) = A(i,j) - gamma*(*w)(i-k-1);
      }
    }
    for(i=0;i<N;i++){
      gamma = 0.0;
      for(q=k+1;q<N;q++)
```

```
        gamma += beta*(*w)(q-k-1)*A(i,q);
      for(j=k+1;j<N;j++){
        A(i,j) = A(i,j) - gamma*(*w)(j-k-1);
      }
    }
  }

  delete x;
  delete w;
}
```

Remark 2: Observe that we pass the reference to the matrix **A** (denoted by the "SCMatrix &A" in the argument list.) We do this because we want to replace the values in **A** with the new upper Hessenberg matrix. If we were to omit the "&" and hence pass by value, the modifications made to the matrix **A** within the function would be lost when the function returns to the calling program.

Remark 3: Inside the function, we dynamically allocate two new SCVectors and assign them to the two pointers w and x. To use the "( )" operator associated with the SCVector class, we first use the unitary operator '*' to retrieve the object to which the pointer points. Hence, the expression (*w) yields the object to which the pointer w points. The extra parentheses around this expression are used to guarantee that the '*' is carried out before the "( )" operator.

9.4 Preconditioned Conjugate Gradient Method - PCGM

9.4.1 Convergence Rate of CGM

We have already introduced the Conjugate Gradient Method (CGM) in section 4.1.7 for the linear system

$$\mathbf{Ax} = \mathbf{b}.$$

By defining the residual of the k^{th} iteration

$$r_{\mathbf{k}} \equiv \mathbf{b} - \mathbf{A}\mathbf{x}_k,$$

the solution and search directions are computed from

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{p}_k && (9.12) \\ \mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k && (9.13) \end{aligned}$$

while the residual can also be computed iteratively, i.e.,

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k. \qquad (9.14)$$

We can derive a three-term recurrence formula by substituting in equation (9.12) the vector $\mathbf{p}_k$ from equation (9.13) and also using the residual definition, to obtain

$$\mathbf{x}_{k+1} = (1+\gamma_k)\mathbf{x}_k + \alpha_k(\mathbf{b} - \mathbf{A}\mathbf{x}_k) - \gamma_k \mathbf{x}_{k-1}$$

where we have defined

$$\gamma_k \equiv \frac{\alpha_k \beta_{k-1}}{\alpha_{k-1}}.$$

This is sometimes referred to as the *Rutishauser* formula. Symmetry and orthogonality together lead to the familiar three-term *magic formula* as we have seen many times in this book!

We have mentioned in section 4.1.7 that the CG method is equivalent to minimizing a properly defined quadratic form. In fact, it can be proved that if $\mathbf{s}$ is the exact solution, then the conjugate gradient iterate $\mathbf{x}_k$ minimizes the norm $||\mathbf{s} - \mathbf{x}||_A$ over the Krylov subspace of dimension k. This space is defined based on powers of $\mathbf{A}$ with the orthogonal directions, i.e.,

$$\mathcal{K}_k(\mathbf{A}, \mathbf{p}) = \text{ span } \{\mathbf{p}, \mathbf{Ap}, \mathbf{A}^2\mathbf{p} \ldots, A^{k-1}\mathbf{p}\}.$$

The Rutishauser formula is similar to the Lanczos three-term formula, see section 10.3.6, and so CGM and Lanczos are related - they both use the same Krylov subspace and have three-term recurrence formulas. More specifically, a tridiagonal matrix $\mathbf{T}_j$ can be constructed for CGM from

$$\mathbf{T}_j = \mathbf{P}^T\mathbf{AP}.$$

Here $\mathbf{P} = \mathbf{RD}^{-1}$, where $\mathbf{D}$ is a diagonal matrix containing the magnitudes of residuals. Also, $\mathbf{R}$ is the product of two matrices, the first formed by the columns of the orthogonal search directions $\mathbf{p}_j$ while the second is a bidiagonal matrix with 1's in the diagonal and the scalars β_j above the main diagonal.

With regards to convergence, Reid [76] has observed that in practice CGM produces very good answers even before the total number of iterations reaches n, where $n \times n$ is the size of the symmetric positive-definite matrix $\mathbf{A}$. We recall that the fundamental theorem on conjugate directions, stated in section 4.1.7, guarantees that the exact solution will be achieved after n iterations. More specifically, if matrix $\mathbf{A}$ has only $m \leq n$ distinct eigenvalues then convergence will be achieved in m iterations. Round-off and corresponding loss of orthogonality is responsible for deviation from the theory, although round-off is not as severe as in the Lanczos method, as we discuss in section 10.3.6.

In practice, we always use a *stopping criterion* for convergence instead of having a loop with n iterations. To this end, it is important that the tolerance ϵ in the convergence test be proportional to the relative reduction of the initial residual or in some cases to be normalized with the right hand-side, e.g.,

$$\| r_{k+1} \|_2 \;\geq\; \epsilon \| b \|_2 \,.$$

Here $\epsilon \approx 10^{-d}$ where d is the number of digits of desired accuracy. The computation of $\| \mathbf{r}_{k+1} \|^2 = (\mathbf{r}_{k+1}, \mathbf{r}_{k+1})$ requires no extra work as this quantity is used in the numerator of the formula for β_k. Also, we note that the residual $\| \mathbf{r}_{k+1} \|_2$ may not be decreasing *monotonically* although the solution error $\| \mathbf{s} - \mathbf{x}_k \|_2$ decreases monotonically. This is because in the minimization procedure the *error* of the solution is targeted directly but not the residual. Direct minimization of the residual is more common in solvers for non-symmetric systems, as we discuss in section 9.5. However, the $\mathbf{A}^{-1}$-residual norm, $(\mathbf{r}^T\mathbf{A}^{-1}\mathbf{r})$, decreases monotonically.

Because of finite arithmetic, the convergence rate of CGM is controlled by the condition number of matrix $\mathbf{A}$, which we denote by $\kappa_2(\mathbf{A})$. Specifically, the following estimate holds, (see [26, 46])

$$\frac{\| \mathbf{s} - \mathbf{x}_k \|_2}{\| \mathbf{s} - \mathbf{x}_0 \|_2} \leq 2\gamma^k \sqrt{\kappa_2(\mathbf{A})}\,,$$

where

$$\kappa_2(\mathbf{A}) = \frac{\lambda_{\max}}{\lambda_{\min}} \text{ and } \gamma = \frac{\sqrt{\kappa_2(\mathbf{A})} - 1}{\sqrt{\kappa_2(\mathbf{A})} + 1}\,.$$

For large values of κ_2 we have that $\gamma \to 1$, and thus the number of iterations for convergence of CGM is proportional to $\sqrt{\kappa_2(\mathbf{A})}$. For example, for the Poisson equation discretized on an N-by-N grid using second-order finite difference discretization (see chapter 6) we have that $\kappa_2(\mathbf{A}) \propto N^2$, and thus the number of iterations is proportional to N. The computational cost of CGM is then similar to SOR assuming for the latter an optimum relaxation parameter.

9.4.2 Preconditioners

In order to accelerate the convergence of CGM we employ preconditioners, see also 7.2.8. That is, we transform the linear system $\mathbf{Ax} = \mathbf{b}$ to

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}$$

by multiplying by the *preconditioner* (non-singular) matrix $\mathbf{M}$. We have already seen in section 7.2.8 some of the desired properties of the preconditioner: It should be:

- Spectrally close to matrix $\mathbf{A}$ so that the condition number

  $$\kappa_2(\mathbf{M}^{-1}\mathbf{A}) \ll \kappa_2(\mathbf{A})$$

  and also,

- Inexpensive to invert since the solution of $\mathbf{Mx} = \mathbf{b}$ will be required.

In addition, since we consider here a symmetric positive-definite matrix $\mathbf{A}$, then $\mathbf{M}$ has to also be symmetric and positive-definite.

The objective is to modify the original CG algorithm only slightly in order to "correct" the search directions but not to increase the computational complexity significantly. To this end, we first need to symmetrize the preconditioned system as $\mathbf{M}^{-1}\mathbf{A}$ is not a symmetric matrix. We then express $\mathbf{M}$ in terms of its eigenvectors and eigenvalues, i.e.,

$$\mathbf{M} = \mathbf{V}\Lambda\mathbf{V}^T \Rightarrow \mathbf{M}^{1/2} \equiv \mathbf{V}\Lambda^{1/2}\mathbf{V}^T.$$

Next we multiply $\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}$ by $\mathbf{M}^{1/2}$ to arrive at

$$(\mathbf{M}^{-1/2}\mathbf{A}\mathbf{M}^{-1/2})\,(\mathbf{M}^{1/2}\mathbf{x}) = \mathbf{M}^{-1/2}\mathbf{b}$$

$$\Rightarrow \mathbf{By} = \mathbf{f}, \quad \text{where} \quad \mathbf{B} \equiv \mathbf{M}^{-1/2}\mathbf{A}\mathbf{M}^{-1/2},$$

and,

$$\mathbf{y} \equiv \mathbf{M}^{1/2}\mathbf{x} \quad \text{and} \quad \mathbf{f} \equiv \mathbf{M}^{-1/2}\mathbf{b}.$$

The above defines the new system (i.e., $\mathbf{B}, \mathbf{y}, \mathbf{f}$) to be solved using the original CGM. We note that

$$\mathbf{M}^{-1/2}\mathbf{B}\mathbf{M}^{1/2} = \mathbf{M}^{-1/2}(\mathbf{M}^{-1/2}\mathbf{A}\mathbf{M}^{-1/2})\mathbf{M}^{1/2} = \mathbf{M}^{-1}\mathbf{A}$$

and thus $\mathbf{B}$ and $\mathbf{M}^{-1}\mathbf{A}$ are similar so they have the same eigenvalues.

The following PCG algorithm is derived by applying the standard CG algorithm to the new system

$$\mathbf{By} = \mathbf{f},$$

as defined above. The important thing is that we do not need to explicitly take the square root of $\mathbf{M}$ – this would have been costly!

Preconditioned Conjugate Gradient Algorithm

```
• Initialize:
    – Choose $\mathbf{x}_0 \Rightarrow \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
    – Solve $\mathbf{M}\tilde{\mathbf{r}}_0 = \mathbf{r}_0 \Rightarrow \mathbf{p}_0 = \tilde{\mathbf{r}}_0$
• Begin Loop: for $k = 1, \ldots$
        $\alpha_k = \frac{(\tilde{\mathbf{r}}_k, \mathbf{r}_k)}{(\mathbf{p}_k, \mathbf{A}\mathbf{p}_k)}$
        $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
        $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$
        $\mathbf{M}\tilde{\mathbf{r}}_{k+1} = \mathbf{r}_{k+1}$
        If $(\tilde{\mathbf{r}}_{k+1}, \mathbf{r}_{k+1}) \le \epsilon$
        If $(\mathbf{r}_{k+1}, \mathbf{r}_{k+1}) \le \epsilon$
        return
        $\beta_k = \frac{(\tilde{\mathbf{r}}_{k+1}, \mathbf{r}_{k+1})}{(\tilde{\mathbf{r}}_k, \mathbf{r}_k)}$
        $\mathbf{p}_{k+1} = \tilde{\mathbf{r}}_{k+1} + \beta_k \mathbf{p}_k$
    endfor
• End Loop
```

Remark 1: The *stopping criterion* is based on the actual residual $\mathbf{r}$ and not on the modified residual $\tilde{\mathbf{r}}$. However, checking the latter first saves some computational time as it is readily available while the former requires explicit calculation.

The question is still open as to what is the *best preconditioner* since in most cases it is problem-dependent. For matrices with diagonal elements that are very different in magnitude, using

$$\mathbf{M} = diag(\, a_{11}, a_{22}, \ldots, a_{nn} \,)$$

is very effective as it reduces the condition number of $\mathbf{B}$ by a factor of n of its minimum value [26]. This is called *diagonal scaling* and corresponds to Jacobi preconditioning. An extension of this idea is to build $\mathbf{M}$ as a block-diagonal matrix out of block submatrices of $\mathbf{A}$. Similarly, the Gauss-Seidel method can be used as preconditioner but it needs to be symmetrized first. To this end, the symmetric SOR (SSOR) we presented in section 7.2.6 can be a more effective block preconditioner.

Diagonal scaling fails if all diagonal elements are equal such as in the matrices resulted from finite difference discretization of diffusion problems on uniforms grids, see chapter 7, or other Toeplitz type matrices (see next section). One of the most effective and popular preconditioners for such problems is based on the **incomplete Cholesky factorization** of the matrix $\mathbf{A}$. We have already presented the Cholesky factorization of $\mathbf{A}$ in section 9.2, which is feasible for symmetric positive-definite matrices, like the ones we consider here. The problem with Cholesky is that it fills-in the zero entries of $\mathbf{A}$ in the LL^T decomposition, and in some cases it may totally destroy the possible sparsity initially present in $\mathbf{A}$.

To obtain an incomplete Cholesky factorization of $\mathbf{A}$ we can simply suppress the fill-in entries, i.e., zeroing out the entries corresponding to the zero entries of the original matrix. Alternatively, to avoid computing these entries in $\mathbf{L}$ and simply place zeros at the corresponding locations, we can modify the Cholesky algorithm so that

- if $a_{ij} \neq 0$, compute l_{ij},
- elseif $l_{ij} = 0$.

We note that the two approaches are not the same as can be easily verified in the following example. Let us consider the (4×4) matrix

$$\mathbf{A} = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} .$$

We first perform a standard Cholesky decomposition, which yields the following triangular matrix

$$\mathbf{L}_c = \begin{bmatrix} 2.0000 & 0 & 0 & 0 \\ -0.5000 & 1.9365 & 0 & 0 \\ -0.5000 & -0.1291 & 1.9322 & 0 \\ 0 & -0.5164 & -0.5521 & 1.8516 \end{bmatrix} .$$

We note that the (3,2) entry which was initially zero in $\mathbf{A}$ has become non-zero in $\mathbf{L}_c$. The first version of incomplete Cholesky would then be:

$$\mathbf{L}_{i1} = \begin{bmatrix} 2.0000 & 0 & 0 & 0 \\ -0.5000 & 1.9365 & 0 & 0 \\ -0.5000 & 0 & 1.9322 & 0 \\ 0 & -0.5164 & -0.5521 & 1.8516 \end{bmatrix} .$$

However, the second version, where we don't compute the l_{ij} at all if the corresponding $a_{ij} = 0$, is

$$\mathbf{L}_{i2} = \begin{bmatrix} 2.0000 & 0 & 0 & 0 \\ -0.5000 & 1.9365 & 0 & 0 \\ -0.5000 & 0 & 1.9365 & 0 \\ 0 & -0.5164 & -0.5164 & 1.8619 \end{bmatrix} ,$$

which is different than $\mathbf{L}_{i1}$ in the entries (3,3), (4,3) and (4,4).

In practice, we can replace the second code statement above by setting a *threshold value* on the Cholesky entry as follows

- elseif $|l_{ij}| \leq \epsilon$ then $l_{ij} = 0$,

so that only significantly large entries are retained. Clearly, the incomplete Cholesky decomposition can be performed only initially, i.e., before the iteration loop, and store $\mathbf{M} = \mathbf{L}\mathbf{L}^T$ or simply store $\mathbf{L}$.

Remark 2: While Cholesky factorization of a symmetric positive-definite matrix is always feasible, incomplete factorization may not be possible in some cases. This complication may manifest itself as a square root of a negative number to compute the l_{ii}. In this case, a positive large number should overwrite the appropriate element, which can be chosen either arbitrarily or as a sum of the adjoint diagonal entries or even as a sum of absolute values of the rest of the entries in the row; the latter will ensure diagonal dominance.

Remark 3: *Domain decomposition* is another way of preconditioning partial differential equations. The idea is to break up the domain of interest to subdomains, which can also overlap, and subsequently solve the PDE approximately but fast in each of the subdomains. This can be also done independently for each domain in an *embarrassingly parallel* fashion. The preconditioner matrix $\mathbf{M}$ is constructed by piecing together the solutions to subproblems leading to a block diagonal $\mathbf{M}$ if the subdomains do not overlap or a product of block diagonal submatrices if the subdomains overlap.

9.4.3 Toeplitz Matrices and Circulant Preconditioners

A special preconditioner is very effective for Toeplitz matrices, the circulant matrix. Topeplitz matrices have constant diagonals and are encountered in signal processing and in a wide range of other problems which are invariant in time and space. A circular matrix $\mathbf{C}$ is a Toeplitz matrix but it is entries also satisfy

$$c_k = c_{k+n}$$

where $n \times n$ is the size of the matrix $\mathbf{C}$.

A general Toeplitz matrix has $(2n-1)$ independent entries, which are determined by the first row and column. A general circulant matrix on the other hand has only n independent entries. Assuming symmetry, then a Toeplitz matrix has n independent entries (the first column) whereas a circulant matrix has $[n/2]+1$. More clearly, the differences between a general Toeplitz and a general circulant matrix are shown below.

$$\mathbf{A} = \begin{bmatrix} a_0 & a_{-1} & . & . & a_{1-n} \\ a_1 & a_0 & a_{-1} & . & . \\ . & a_1 & a_0 & . & . \\ . & . & . & . & a_{-1} \\ a_{n-1} & . & . & a_1 & a_0 \end{bmatrix},$$

and

$$\mathbf{C} = \begin{bmatrix} c_0 & c_{n-1} & \cdot & \cdot & c_1 \\ c_1 & c_0 & c_{n-1} & \cdot & \cdot \\ \cdot & c_1 & c_0 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & c_{n-1} \\ c_{n-1} & \cdot & \cdot & c_1 & c_0 \end{bmatrix}.$$

Assuming we want to solve the system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a *symmetric Toeplitz* matrix, the suggestion, first proposed in [83], is to precondition it with a corresponding circulant matrix. To this end, we recall that in most applications the main diagonal and its neighbors are strongly dominant and thus we can use them to construct an appropriate circulant preconditioner $\mathbf{C}$. The entries $c_1 = a_1$ appear both next to the main diagonal but also in the extreme corners as shown above. It turns out that these relatively large entries control the eigenspectrum of $\mathbf{C}^{-1}\mathbf{A}$, which in turn controls the convergence rate of PCGM. The rest of the eigenvalues are clustered around one; this is demonstrated in homework problems, see 10.7. Typically, we copied a few of the main diagonals from the Toeplitz matrix onto the circulant matrix but of course not all! In the homework problems of section 9.8 we ask you to experiment with different circulant preconditioners. As you can see, the speed-up is substantial!

9.4.4 Parallel PCGM

The parallelization of the preconditioned conjugate gradient is fairly straightforward. Assuming that the matrix $\mathbf{A}$ has been distributed by rows across processes, the conjugate gradient component of the algorithm can be accomplished with only four MPI calls: three calls to *MPI_Allreduce* to accomplish *dot products*, and one call to *MPI_Allgather*. We can eliminate one of the reduction calls if we decide to use the *dot product* of the residual with the modified residual for the stopping criterion. Depending on the choice of preconditioner, however, additional MPI calls may be required to accomplish the preconditioning. In the case of a diagonal preconditioner, no MPI calls are necessary since diagonal preconditioning can be accomplished locally on all rows contained in a process. Other preconditioners such as incomplete Cholesky may require additional MPI calls, the cost of which should be considered when determining what preconditioner to use. In figure 9.14 we provide a schematic of the iterative part of the parallel PCG algorithm with annotations denoting what BLAS and MPI operations should be used. Note that in figure 9.14 we use the modified residual for the stopping criterion, and hence we only have three MPI calls.

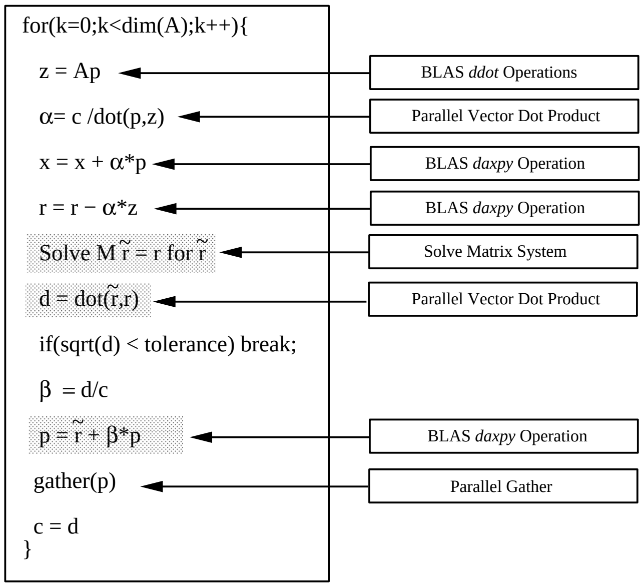

Figure 9.14: Iterative part of the preconditioned conjugate gradient (PCG) algorithm.

Software ⊙ *Suite*

We now present an MPI program demonstrating the preconditioned conjugate gradient method with a diagonal preconditioner. As a sample problem, we will solve (using second-order finite differences) the following equation:

$$\frac{d^2u(x)}{dx^2} - c_2 e^{c_1(x-0.5)^2} u(x) = -sin(2\pi x)(4\pi^2 - c_2 e^{c_1(x-0.5)^2}) \qquad (9.15)$$

in the interval $x \in [0,1]$ with boundary conditions $u(0) = u(1) = 0$ and constants $c_1 = 20.0$ and $c_2 = 1000.0$. The exact solution is $u(x) = sin(2\pi x)$.

We have chosen the constants c_1 and c_2 so that there is a large disparity in the values along the diagonal. This will allow us to see the difference between conjugate gradient (CGM) and preconditioned conjugate gradient (PCGM) with a diagonal preconditioner.

Since we have chosen to use a second-order finite difference method to approximate the derivative operator, we expect that the error should decrease by a factor of four when we double the number of grid points used. In figure 9.15 we show a log-log plot of the L_2 error versus the number of grid points used. The magnitude of the slope of the line is approximately two, consistent with the fact that our approximation is second order.

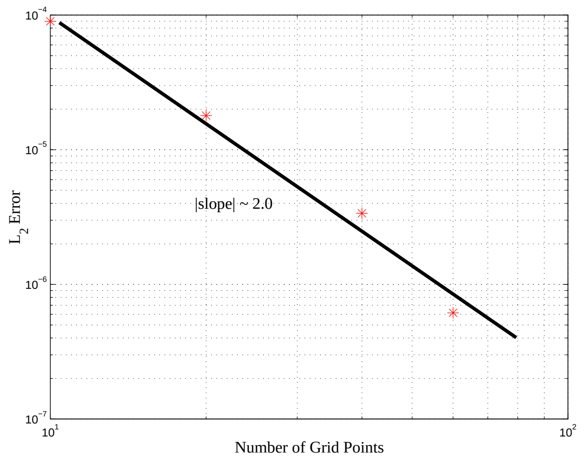

Figure 9.15: L_2 error versus the number of grid points for the PCGM example defined in equation (9.15).

In figure 9.16 we plot the inner product of the residual (r, r) for both the CGM and the PCGM. As predicted, PCGM converges much faster than the standard CGM.

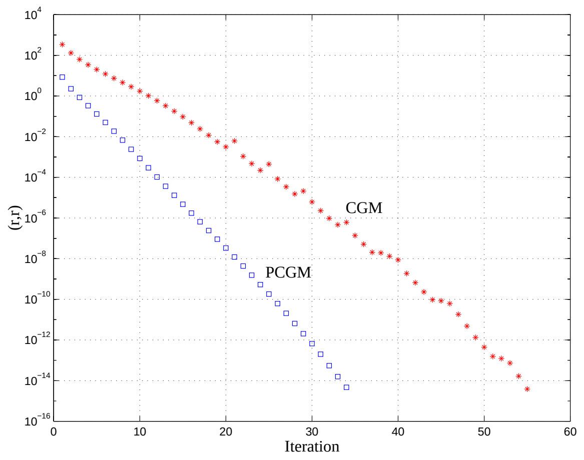

Figure 9.16: Inner product of the residual for both the CG and PCG methods applied to the PCGM example defined in equation (9.15).

To better explain the code, we have broken the entire program into four parts, labeled part one through part four. The four parts are as follows:

1. Part 1 - MPI initialization, initial memory allocation, and generation of grid and right-hand-side vector.

2. Part 2 - Memory allocation and generation of the matrix **A**.
3. Part 3 - PCGM initialization.
4. Part 4 - PCGM main iteration loop.

For each part, we will first present the code and then present a collection of remarks elucidating the salient points within each part.

Part 1 - MPI initialization

```
#include <iostream.h>
#include <iomanip.h>
#include "SCmathlib.h"
#include<mpi.h>

const int rows_per_proc = 40;
const double c1 = 20.0;
const double c2 = 1000.0;
const double tol = 1.0e-14;

int main(int argc, char *argv[]){
  int i,j,k;
  int mynode, totalnodes, totalsize, offset;
  MPI_Status status;
  double sum,local_sum,c,d,alpha,beta;
  double ** A, *q, *x, *grid;

  double *p,*z,*r,*mr;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  totalsize = totalnodes*rows_per_proc;

  p = new double[totalsize];
  z = new double[rows_per_proc];
  r = new double[rows_per_proc];
  mr = new double[rows_per_proc];
  x = new double[rows_per_proc];
  q = new double[rows_per_proc];
  grid = new double[rows_per_proc];

  double dx = 1.0/(totalsize+1);
  for(i=0;i<rows_per_proc;i++){
    grid[i] = dx*(1+rows_per_proc*mynode+i);
    q[i] = -dx*dx*sin(2.0*M_PI*grid[i])*
      (-4.0*M_PI*M_PI - c2*exp(c1*(grid[i]-0.5)*(grid[i]-0.5)));
    x[i] = 1.0;
  }
```

Remark 1: We have four global variables in this program. The variable *rows_per_proc* gives the number of rows per processor. In this program we have decomposed the matrix by associating rows to processors. The two constants *c1* and *c2* are specific to the problem we are solving. The last global constant variable *tol* specifies the tolerance to which we should converge.

Remark 2: Observe that with the exception of the array *p* all other arrays need only be of size *rows_per_proc* and not of size *totalsize*. Each processor need only maintain its part of the residual, modified residual and solution vector; however, each processor must have a copy of the entire *p* vector.

Part 2 - Memory allocation and generation of matrix

```
  /* Part 2 */

  A = new double*[rows_per_proc];
  for(i=0;i<rows_per_proc;i++){
    A[i] = new double[totalsize];
    for(j=0;j<totalsize;j++)
      A[i][j] = 0.0;
  }

  if(mynode==0){
    A[0][0] = 2.0 + dx*dx*c2*exp(c1*(grid[0]-0.5)*(grid[0]-0.5));
    A[0][1] = -1.0;
    for(i=1;i<rows_per_proc;i++){
      A[i][i] = 2.0 + dx*dx*c2*exp(c1*(grid[i]-0.5)*
                (grid[i]-0.5));
      A[i][i-1] = -1.0;
      A[i][i+1] = -1.0;
    }
  }
  else if(mynode == (totalnodes-1)){
    A[rows_per_proc-1][totalsize-1] = 2.0 +
        dx*dx*c2*exp(c1*(grid[rows_per_proc-1]-0.5)*
        (grid[rows_per_proc-1]-0.5));
    A[rows_per_proc-1][totalsize-2] = -1.0;
    for(i=0;i<rows_per_proc-1;i++){
      offset = rows_per_proc*mynode + i;
      A[i][offset] = 2.0 + dx*dx*c2*exp(c1*(grid[i]-0.5)*
                     (grid[i]-0.5));
      A[i][offset-1] = -1.0;
      A[i][offset+1] = -1.0;
    }
  }
  else{
    for(i=0;i<rows_per_proc;i++){
      offset = rows_per_proc*mynode + i;
      A[i][offset] = 2.0 + dx*dx*c2*exp(c1*(grid[i]-0.5)*(grid[i]-0.5));
      A[i][offset-1] = -1.0;
      A[i][offset+1] = -1.0;
    }
  }
```

Remark 3: We break the matrix setup into three cases. We have to carefully handle the first and last processors because they contain the first and last rows, respectively.

$\boxed{\text{Part 3 - PCGM initialization}}$

```
  /* Part 3 */

  offset = mynode*rows_per_proc;

  for(i=0;i<totalsize;i++)
    p[i] = 1.0;

  for(i=0;i<rows_per_proc;i++){
    r[i] = q[i] - dot(totalsize,A[i],p); //calculation of residual
    mr[i] = r[i]/A[i][offset+i];   //calculation of modified residual
  }

  local_sum = dot(rows_per_proc,mr,r);
  MPI_Allreduce(&local_sum,&sum,1,MPI_DOUBLE,MPI_SUM,
                MPI_COMM_WORLD);
  c = sum;

  MPI_Allgather(mr,rows_per_proc,MPI_DOUBLE,p,rows_per_proc,
                MPI_DOUBLE,MPI_COMM_WORLD);
```

Remark 4: We have chosen our initial vector as the vector of all ones. Because we have to calculate the initial residual, we require that all processors have the entire initial guess. Notice that we temporarily use the p array to accomplish this instead of allocating a new vector. Because p is not in use until after the modified residual is calculated, we can use the allocated space with no adverse effect.

Part 4 - PCGM main iteration loop

```
 /* Part 4 */

  for(k=0;k<totalsize;k++){

    for(i=0;i<rows_per_proc;i++)
      z[i] = dot(totalsize,A[i],p);

    local_sum = dot(rows_per_proc,z,p+offset);

    MPI_Allreduce(&local_sum,&sum,1,MPI_DOUBLE,MPI_SUM,
                  MPI_COMM_WORLD);

    alpha = c/sum;

    for(i=0;i<rows_per_proc;i++){
      x[i] = x[i] + alpha*p[offset+i];
      r[i] = r[i] - alpha*z[i];
    }
    /* Preconditioning Stage */
    for(i=0;i<rows_per_proc;i++)
      mr[i] = r[i]/A[i][offset+i];

    local_sum = dot(rows_per_proc,mr,r);

    MPI_Allreduce(&local_sum,&sum,1,MPI_DOUBLE,MPI_SUM,
                  MPI_COMM_WORLD);

    d = sum; //contains inner product of
             //residual and modified residual

    local_sum = dot(rows_per_proc,r,r);

    MPI_Allreduce(&local_sum,&sum,1,MPI_DOUBLE,MPI_SUM,
                  MPI_COMM_WORLD);

    //sum now contains inner product of residual and residual

    if(mynode == 0){
      cout << k << "\t" << "dot(mr,r) = " << d << "\t";
      cout << "dot(r,r) = " << sum << endl;
    }

    if(fabs(d) < tol) break;
    if(fabs(sum) < tol) break;

    beta = d/c;

    for(i=0;i<rows_per_proc;i++)
      z[i] = mr[i] + beta*p[i+offset];

    MPI_Allgather(z,rows_per_proc,MPI_DOUBLE,p,rows_per_proc,
                  MPI_DOUBLE,MPI_COMM_WORLD);

    c = d;

  }

  delete[] p;
  delete[] z;
  delete[] r;
  delete[] mr;
  delete[] x;
  delete[] q;
  delete[] grid;
  for(i=0;i<rows_per_proc;i++)
    delete[] A[i];
  delete[] A;

  MPI_Finalize();
}
```

Remark 5: Observe that we only require four MPI calls: three *MPI_Allreduce* and one *MPI_Allgather*. The three reductions are used to obtain the inner product across all processors; the *Allgather* is used so that all processors have the updated value of p.

Remark 6: Because we are using diagonal preconditioning, no additional communication is required for the preconditioner. If we were to use incomplete Cholesky as a preconditioner, additional communication similar to what was accomplished in the parallel Gaussian elimination solver would have to be used. The factorization would be accomplished before the iteration loop, and only the cost of two parallel back solves would be incurred within the iteration loop.

9.5 Non-Symmetric Systems

The conjugate gradient algorithm assumes that the matrix $\mathbf{A}$ is symmetric and positive-definite. To deal with the non-symmetric system

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

we need to come up with new solvers or transform the non-symmetric system into a symmetric one as follows

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b} \quad \text{(CGNR)}\,.$$

Clearly, this transformation produces a symmetric positive-definite matrix $\mathbf{B} \equiv \mathbf{A}^T\mathbf{A}$ and thus we can apply PCGM to it. This is called the CGNR (conjugate gradient normal residual) algorithm and it is an acceptable method for matrices which are well-conditioned, although we have to pay extra to include matrix-vector multiplies with both $\mathbf{A}^T$ and $\mathbf{A}$. On the other hand, the condition number of $\mathbf{B}$ is the *square* of the condition number of $\mathbf{A}$ and thus for ill-conditioned matrices an extremely large number of iterations is required for convergence.

We now compare the CG and CGNR algorithms by considering what quantity is that we *minimize* in these minimization-based solvers. We have already seen in section 4.1.7 that the solution with CG is equivalent to minimizing the quadratic form

$$P_{CG}(\mathbf{x}) = \frac{1}{2}(\mathbf{x}, \mathbf{A}\mathbf{x}) - (\mathbf{b}, \mathbf{x}).$$

We can re-write the above form in terms of the error $\mathbf{e} \equiv \mathbf{x} - \mathbf{s}$, where $\mathbf{s}$ is the exact solution, as follows

$$\begin{aligned} 2P_{CG}(\mathbf{x}) &= (\mathbf{x}, \mathbf{A}\mathbf{x}) - 2(\mathbf{x}, \mathbf{A}\mathbf{s}) + (\mathbf{s}, \mathbf{A}\mathbf{s}) \\ &= \mathbf{e}^T\mathbf{A}\mathbf{e} \\ &= ||\mathbf{e}||_A. \end{aligned}$$

The above simply states that searching for the solution using the CG algorithm is equivalent to minimizing the *error* in the A-norm. In contrast, the CGNR algorithm minimizes the residual in the L_2-norm, and thus the corresponding quadratic form is

$$\begin{aligned} 2P_{CGNR}(\mathbf{x}) &= ||\mathbf{b} - \mathbf{A}\mathbf{x}||_2 \\ &= (\mathbf{b} - \mathbf{A}\mathbf{x}, \mathbf{b} - \mathbf{A}\mathbf{x}) \\ &= (\mathbf{A}\mathbf{x}, \mathbf{A}\mathbf{x}) - 2(\mathbf{b}, \mathbf{A}\mathbf{x}) + (\mathbf{b}, \mathbf{b}). \end{aligned}$$

Therefore, the minimization of P_{CGNR} corresponds to applying CG to the system

$$\mathbf{B}\mathbf{x} = \mathbf{f}$$

where $\mathbf{B} = \mathbf{A}^T\mathbf{A}$ and $\mathbf{f} = \mathbf{A}^T\mathbf{b}$.

There are several *conjugate residual* algorithms in the literature, which are based on the minimization of the residual in different forms. A significant difference between such algorithms and the CG for symmetric systems is that the loss of symmetry results in the loss of the three-term magic recurrence formula, which keeps things sparse and leads to efficiency both in memory and cost. Instead of two or three vectors, in the non-symmetric conjugate algorithms we typically need to store the entire sequence of conjugate directions, which is prohibitively expensive for very large n. However, in practice a subset of the last k vectors may work or a *restart* of the algorithm after k steps can also be used.

In the following, instead of presenting all variants of conjugate gradient residual algorithms we concentrate on one that makes use of an iterative procedure to simplify the original matrix, namely the *Arnoldi iteration.* Based on this procedure, we can solve, relatively efficiently, non-symmetric systems as we demonstrate below. We can also compute eigenvalues of non-symmetric matrices, as we discuss in section 10.5 in the next chapter.

9.5.1 The Arnoldi Iteration

The Arnoldi method is an orthogonal projection onto a Krylov subspace $\mathcal{K}_m$ for non-symmetric matrices $\mathbf{A}(n \times n)$; usually $m \ll n$. It reduces the matrix $\mathbf{A}$ to a *Hessenberg form*, that is it accomplishes what Householder does for the entire matrix $\mathbf{A}$ but here we have the option of only going half-way. Arnoldi, who introduced this algorithm in the early 1950s [3], suggested that it leads to good approximations for some of the eigenvalues of $\mathbf{A}$ even if we terminate prematurely! In practice, it is a useful technique for obtaining the eigenvalues of *large sparse* matrices. It is useful to think of this as an extension of Lanczos method (see section 10.3.6) to non-symmetric matrices. It has the same iterative and approximate form - although a separate version of Lanczos exists for non-symmetric matrices [79].

Let us begin by considering the similarity transformation

$$\mathbf{A} = \mathbf{V}^T \mathbf{H} \mathbf{V},$$

where we define the matrix $\mathbf{V}$ or $\mathbf{V}_m$ with orthonormal columns

$$\mathbf{V}_m = [\,\mathbf{v}_1 \,|\, \mathbf{v}_2 \,|\, \ldots \,|\, \mathbf{v}_m\,],$$

and $\mathbf{H}_m$ is the Hessenberg matrix $(m \times m)$. We also define the extended matrix $\tilde{\mathbf{H}}_m$ of dimension $(m+1) \times m$, as follows:

$$\tilde{\mathbf{H}}_m = \begin{bmatrix} h_{11} & h_{12} & \ldots & h_{1m} \\ h_{21} & h_{22} & \ldots & h_{2m} \\ & \ddots & \ddots & \vdots \\ & & h_{m,m-1} & h_{mm} \\ & & & h_{m+1,m} \end{bmatrix},$$

and we can write

$$\tilde{\mathbf{H}}_m = \mathbf{H}_m + h_{m+1,m}\mathbf{v}_{m+1}\mathbf{e}_m^T,$$

where $\mathbf{v}_{m+1}$, defined as orthonormal, i.e., it has unit norm.

The Arnoldi iteration process satisfies:

$$\mathbf{A}\mathbf{V}_m = \mathbf{V}_{m+1}\tilde{\mathbf{H}}_m\,, \tag{9.16}$$

and correspondingly, the m^{th} column of this equation reads:

$$\mathbf{A}\mathbf{v}_m = h_{1m}\mathbf{v}_1 + h_{2m}\mathbf{v}_2 + \ldots + h_{mm}\mathbf{v}_m + h_{m+1,m}\mathbf{v}_{m+1},$$

which is an $(m+1)$-term recurrence formula, instead of the three-term formula in symmetric systems.

The basic algorithm of Arnoldi iteration implements the above formula in a straightforward manner, as follows:

Basic Arnoldi Algorithm

```
Initialize: Choose a vector $\mathbf{x}_0 \Rightarrow \mathbf{v}_1 = \frac{\mathbf{x}_0}{||\mathbf{x}_0||}$
    Begin Loop: for $j = 1, 2, \ldots m$
                for $i = 1, \ldots, j$
                    $h_{ij} = (\mathbf{A}\mathbf{v}_j, \mathbf{v}_i)$
                endfor
                $\mathbf{w}_j = \mathbf{A}\mathbf{v}_j - \sum_{i=1}^{j} h_{ij}\mathbf{v}_i$
                $h_{j+1,j} = \| \mathbf{w}_j \|_2$
                $\mathbf{v}_{j+1} = \mathbf{w}_j / h_{j+1,j}$
    End Loop:   endfor
```

The above algorithm is basically the standard Gram-Schmidt orthogonalization procedure applied to the Krylov space $\mathcal{K}_m$.

We note that in each iteration we compute the entire j column, including the entry $h_{j+1,j}$ below the main diagonal. In particular, if $h_{j+1,j} = 0$ then the Arnoldi iteration stalls, but that is in fact good news! This is because such a breakdown usually implies that we have achieved convergence and the iteration should be terminated. If we want to continue, we can *restart* the process with a new orthonormal vector $\mathbf{v}_{j+1}$, which can be selected arbitrarily.

The implementation of the Arnoldi algorithm given above although straightforward it suffers from round-off errors, just as is the case of the basic Gram-Schmidt algorithm. To this end, we can apply the *more stable* modified Gram-Schmidt orthogonalization procedure to the Krylov space to come up with a better code for the Arnoldi iteration, as follows:

Modified Arnoldi Algorithm

```
Initialize: Choose $\mathbf{x}_0 \Rightarrow \mathbf{v}_1 = \frac{\mathbf{x}_0}{\|\mathbf{x}_0\|_2}$.

Begin Loop:  for $j = 1, \ldots, m$
                 $\mathbf{w} = \mathbf{A}\mathbf{v}_j$
                 for $i = 1, \ldots, j$
                     $h_{ij} = (\mathbf{w}, \mathbf{v}_i)$
                     $\mathbf{w} = \mathbf{w} - h_{ij}\mathbf{v}_i$
                 endfor
                 $h_{j+1,j} = \| \mathbf{w} \|_2$
                 $\mathbf{v}_{j+1} = \mathbf{w}/h_{j+1,j}$
End Loop:    endfor
```

This is a much more stable algorithm but even this version needs further treatment sometimes, so an extra orthogonalization may be required occasionally. For robustness, we can apply the Householder algorithm periodically for extra orthogonalization.

Software Suite *Putting it into Practice*

Below we present a *function* for computing the Arnoldi decomposition using the *modified algorithm* presented earlier. The function takes four arguments: the integer value m denoting the dimension of the Krylov space on to which we are projecting, a SCMatrix A of size $N \times N$ on which the decomposition is to be accomplished, and two SCMatrix variables H and V for storing the resulting $\tilde{\mathbf{H}}$ and $\tilde{\mathbf{V}}$ from the decomposition. Note that H must be a SCMatrix of size $(m+1) \times m$ and V must be a SCMatrix of size $N \times (m+1)$. The initial direction vector x defaults to the first unit vector.

```
void ModifiedArnoldi(int m, const SCMatrix &A, SCMatrix &H,
                     SCMatrix &V){
  SCVector v(A.Rows()),w(A.Rows());
  v.Initialize(0.0);
  v(0) = 1.0;

  V.PutColumn(0,v);

  for(int j=0;j<m;j++){
    w = A*v;
    for(int i=0;i<=j;i++){
      V.GetColumn(i,v);
      H(i,j) = dot(w,v);
      w = w - H(i,j)*v;
    }
    H(j+1,j) = w.Norm_l2();
    v = w/H(j+1,j);
    V.PutColumn(j+1,v);
  }
}
```

Remark 1: Once again we pass all three SCMatrix variables by reference (denoted by the '&' symbol) as opposed to by the default case of passing by value. For the variables H and V, the reason is as before - we want to change the values within the matrices, and we want those changes to remain valid after the function has returned (i.e., not be lost in the "pass by value" copy of the variable discarded when the function returns). However, in the case of the matrix A, this is not the case. We do not want to modify the values of A. Why then do we pass by reference as opposed to passing by value? In this case, we do so because we assume that A is very large, and therefore we do not wish to allocate space for a copy of A when the function is called. We instead pass the matrix by reference so that no additional memory must be allocated; the original memory from the calling function is used. We do, however, want to guarantee that the matrix A does not change within the function, hence why we add the **const** in the appropriate place within the argument list. The "const SCMatrix &A" allows us to pass the matrix A by reference, but not to update its values within the function *ModifiedArnoldi*.

Remark 2: We introduced two new SCMatrix methods:

1. SCMatrix::PutColumn(int col, const SCVector &v), and
2. SCMatrix::GetColumn(int col, SCVector &v).

The first function copies the contents of the vector v into the *col* column of the matrix. The second function copies the contents of the *col* column from the matrix into the vector v.

Example: As an example of the use of the code above, we revisit our old friend the Hilbert matrix of size three:

$$\mathbf{A} = \begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix}.$$

The goal is to find the decomposition when $m = 2$. We input into our function $m = 2$, the Hilbert matrix $\mathbf{A}$ as given above, and two SCMatrix variables H and V which are of size 3×2 and 3×3 respectively. As output we obtain $\tilde{\mathbf{H}}$ and $\tilde{\mathbf{V}}$ contained within the SCmatrix variables H and V respectively. The decomposition for the 3×3 Hilbert matrix given above is:

$$\tilde{\mathbf{V}} = \begin{bmatrix} 1.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.8321 & -0.5547 \\ 0.0000 & 0.5547 & 0.83205 \end{bmatrix}$$

and

$$\tilde{\mathbf{H}} = \begin{bmatrix} 1.00000 & 0.60093 \\ 0.60093 & 0.52308 \\ 0.00000 & 0.03462 \end{bmatrix}.$$

9.5.2 GMRES

GMRES stands for "generalized minimum residual" and is one of the most effective solvers for non-symmetric systems

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

where $\mathbf{A}$ is an $n \times n$ square but non-symmetric matrix; it is also assumed non-singular since we look for the solution $\mathbf{A}^{-1}\mathbf{b}$.

The main idea of GMRES is to minimize the residual $\| \mathbf{b} - \mathbf{A}\mathbf{x}_m \|_2$ at the m^{th} iteration. Specifically, $\mathbf{x}_m$ is a vector in the Krylov space

$$\mathcal{K}_m = \{\mathbf{v}, \mathbf{A}\mathbf{b}, \ldots, \mathbf{A}^{m-1}\mathbf{b}\}$$

and it can be determined by solving a least-squares problem; here $m < n$. At the point $m = n$ we are attempting to minimize the residual $\| \mathbf{b} - \mathbf{A}\mathbf{x}_n \|_2$ and hence obtain the solution $\mathbf{x}$.

Such a minimization is equivalent to performing a QR decomposition to the matrix of *least-squares coefficients* $\mathbf{C}$ (see section 3.1.7). We illustrate this point next.

Let us assume that in matrix form we seek to find

$$\| \mathbf{C}\mathbf{a} - \mathbf{f} \|_2 = \text{minimum}$$

with respect to $\mathbf{a}$. Then, we need to QR-decompose $\mathbf{C}$ which is a non-square matrix $p \times q$, $p > q$.

The idea is to apply Householder transformation to the extended matrix $\mathbf{C}$ of $p \times p$ but stop when we have completed zeroing out the entries in the first q columns. We then write:

$$\mathbf{C} = \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix},$$

where $\mathbf{R}$ is a $q \times q$ upper triangular matrix. Since $\| \mathbf{Q} \| = \| \mathbf{Q}^{-1} \| = 1$, we also have

$$\| \mathbf{C}\mathbf{a} - \mathbf{f} \|_2 = \| \mathbf{Q}^{-1}(\mathbf{C}\mathbf{a} - \mathbf{f}) \|_2 = \| \tilde{\mathbf{R}}\mathbf{a} - \tilde{\mathbf{f}} \|_2,$$

where we have defined $\tilde{\mathbf{R}} = \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix}$ and also $\tilde{\mathbf{f}} = \mathbf{Q}^{-1}\mathbf{f}$.

But

$$\tilde{\mathbf{R}}\mathbf{a} - \tilde{\mathbf{f}} = \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix} \mathbf{a} - \begin{bmatrix} \tilde{\mathbf{f}}_1 \\ \tilde{\mathbf{f}}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{R}\mathbf{a} - \tilde{\mathbf{f}}_1 \\ -\tilde{\mathbf{f}}_2 \end{bmatrix}$$

where we split $\tilde{\mathbf{f}}$ in $\tilde{\mathbf{f}}_1$ of length q and $\tilde{\mathbf{f}}_2$ of length $(p - q)$. Then,

$$\| \mathbf{C}\mathbf{a} - \mathbf{f} \|_2^2 = \| \mathbf{R}\mathbf{a} - \tilde{\mathbf{f}}_1 \|_2^2 + \| \tilde{\mathbf{f}}_2 \|_2^2 .$$

Therefore, the least-squares problem is equivalent to solving

$$\mathbf{R}\mathbf{a} = \tilde{\mathbf{f}}_1,$$

which minimizes the entire residual, since $\tilde{\mathbf{f}}_2$ is fixed with respect to $\mathbf{a}$.

The problem, however, in applying directly this QR procedure to

$$\| \mathbf{A}\mathbf{x}_m - \mathbf{b} \|_2 = \text{ minimum}$$

involved in GMRES is that it leads to numerical instabilities. This is exactly where the Arnoldi iteration comes in! The idea is to replace

$$\mathbf{x}_m = \mathbf{V}_m \mathbf{y}$$

and seek to find

$$\| \mathbf{A}\mathbf{V}_m \mathbf{y} - \mathbf{b} \|_2 = \text{ minimum.}$$

Here $\mathbf{V}_m$ is the matrix containing as columns the orthonormal vectors produced by the Arnoldi iteration. Notice that we still use the Krylov iteration but we basically orthonormalize them first via Arnoldi.

We can simplify the above problem since $\mathbf{A}\mathbf{V}_m = \mathbf{V}_{m+1}\tilde{\mathbf{H}}_m$, where $\tilde{\mathbf{H}}_m$ is the extended Hessenberg matrix we have encountered before. Therefore, we seek to find

$$\| \mathbf{V}_{m+1}\tilde{\mathbf{H}}_m \mathbf{y} - \mathbf{b} \|_2 = \text{ minimum}$$

and after multiplying by $\mathbf{V}^T_{m+1}$ (whose norm is unity), we have

$$\| \tilde{\mathbf{H}}_m \mathbf{y} - \mathbf{V}^T_{m+1}\mathbf{b} \|_2 = \text{ minimum.}$$

Finally, since $\mathbf{V}^T_{m+1}\mathbf{b} = \| \mathbf{b} \|_2 \, \mathbf{e}_1$ by construction of the first orthogonal vector $\mathbf{v}_1$, the minimization problem is:

$$\| \tilde{\mathbf{H}}_m \mathbf{y} - \| \mathbf{b} \|_2 \, \mathbf{e}_1 \|_2 = \text{ minimum.}$$

Applying QR to this problem now involves the Hessenberg matrix $\tilde{\mathbf{H}}_m$, so

$$\mathbf{Q}_m \tilde{\mathbf{H}}_m = \tilde{\mathbf{R}}_m = \begin{bmatrix} \mathbf{R}_m \\ 0 \end{bmatrix},$$

where $\mathbf{R}_m$ is an $m \times m$ upper triangular matrix, and $\mathbf{Q}_m$ is an $(m+1) \times (m+1)$ orthogonal matrix.

We can exploit the structure of the Hessenberg matrix and use Givens rotations for accomplishing the decomposition. Recall that the Hessenberg matrix contains an off-diagonal below the main diagonal. This set of entries can easily by modified to zero by the appropriate Givens rotations yielding an upper-triangular matrix $\mathbf{R}$.

Following the discussion above on least-squares, we can compute $\mathbf{y}_m$ from

$$\mathbf{R}_m\mathbf{y} = \parallel \mathbf{b} \parallel \mathbf{q}_1 \,,$$

where $\mathbf{q}_1$ is the *first column* of $\mathbf{Q}_m$ excluding the last entry, i.e., $\mathbf{q}_1$ is a vector of length m. Finally, we compute $\mathbf{x}_m$ from

$$\mathbf{x}_m = \mathbf{V}_m\mathbf{y}.$$

For convergence purposes we need to compute the residual $\mathbf{r}_m$. This is done efficiently as follows:

$$\begin{aligned}
\parallel \mathbf{r}_m \parallel_2 &= \parallel \tilde{\mathbf{H}}_m\mathbf{y} - \parallel \mathbf{b} \parallel_2 \mathbf{e}_1 \parallel_2 \\
&= \parallel \mathbf{Q}_m(\tilde{\mathbf{H}}_m\mathbf{y} - \parallel \mathbf{b} \parallel_2 \mathbf{e}_1) \parallel_2 \\
&= \parallel \mathbf{R}_m\mathbf{y} - \parallel \mathbf{b} \parallel \tilde{\mathbf{q}}_1 \parallel_2 \\
&= \parallel \underbrace{(\mathbf{R}_m y - \parallel \mathbf{b} \parallel \mathbf{q}_1)}_{0} + \parallel \mathbf{b} \parallel (\mathbf{q}_1 - \tilde{\mathbf{q}}_1) \parallel_2 \\
&= \parallel \mathbf{b} \parallel \times (\text{last entry of} |\tilde{\mathbf{q}}_1|) \,,
\end{aligned}$$

where $\tilde{\mathbf{q}}_1$ is the extended $(m+1)$ vector produced by Arnoldi. This formula then gives a very inexpensive way to compute the residual and check convergence.

Remark 1: Unlike CGM, in GMRES the residuals decrease monotonically, i.e.,

$$\parallel \mathbf{r}_{m+1} \parallel_2 < \parallel \mathbf{r}_m \parallel_2$$

because the corresponding Krylov spaces are nested and the *residual* is minimized directly. We recall that in CGM the *error* is minimized instead, and thus reduction of the residual in the L_2-norm is not guaranteed.

Remark 2: From Arnoldi, we can check the value of $h_{m+1,m}$ to tell us when the iteration can be terminated. When the condition $h_{m+1,m} = 0$ is satisfied, we know that the solution $\mathbf{x}$ lies within the Krylov space $\mathcal{K}_m$. This being the case, the least-squares problem over the Krylov space will produce the exact solution.

Example: We now present an example to demonstrate the GMRES process. Consider the following matrix

$$\mathbf{A} = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 2.0 & 5.0 & 7.0 \\ 3.0 & 8.0 & 9.0 \end{bmatrix}$$

and right-hand-side vector

$$\mathbf{b} = \begin{bmatrix} 0.0 \\ 1.0 \\ 2.0 \end{bmatrix}.$$

We will assume that our initial guess $\mathbf{x}_0 = \mathbf{0}$. Using our modified Arnoldi solver with $m = 3$ we obtain the extended Hessenberg matrix

$$\tilde{\mathbf{H}}_m = \begin{bmatrix} 14.2000 & 4.8652 & -0.8305 \\ 4.3081 & 1.0414 & 0.3855 \\ 0.0000 & 0.3855 & -0.2414 \\ 0.0000 & 0.0000 & 0.0000 \end{bmatrix}$$

and corresponding matrix

$$\mathbf{V}_{m+1} = \begin{bmatrix} 0.0000 & 0.8305 & 0.5571 & 0.4082 \\ 0.4472 & 0.4983 & -0.7428 & 0.8165 \\ 0.8944 & -0.2491 & 0.3714 & 0.4082 \end{bmatrix}.$$

Both of these matrices will be used. A linear combination of the first three columns of $\mathbf{V}_{m+1}$ will be used to form the solution. To determine what the proper combination is, we now want to solve the following problem:

$$\| \tilde{\mathbf{H}}_m \mathbf{y} - \| \mathbf{b} \|_2 \, \mathbf{e}_1 \|_2 = \text{ minimum.}$$

where $\tilde{\mathbf{H}}_m$ is the matrix above and $\| \mathbf{b} \|_2 \, \mathbf{e}_1 = (\sqrt{5} \; 0 \; 0 \; 0)^T$.

To solve the minimization problem, we accomplish QR decomposition using Givens rotations. For this problem, the three rotation matrices $\mathbf{G}_0$, $\mathbf{G}_1$ and $\mathbf{G}_2$ are given below:

$$\mathbf{G}_0 = \begin{bmatrix} 0.0645 & 0.0196 & 0 & 0 \\ -0.0196 & 0.0645 & 0 & 0 \\ 0 & 0 & 1.0000 & 0 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}$$

$$\mathbf{G}_1 = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -0.1876 & 2.5802 & 0.0 \\ 0.0 & -2.5802 & -0.1876 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0000 \end{bmatrix}$$

$$\mathbf{G}_2 = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -16.4509 & 0.0 \\ 0.0 & 0.0 & 0.0 & -16.4509 \end{bmatrix}.$$

Each successive Givens rotation matrix is formed based upon the updated matrix. Hence, $\mathbf{G}_0$ is formed based upon the entries of $\tilde{\mathbf{H}}_m$, $\mathbf{G}_1$ is formed based upon the entries of $\mathbf{G}_0\tilde{\mathbf{H}}_m$, and $\mathbf{G}_2$ is formed based upon the entries of $\mathbf{G}_1\mathbf{G}_0\tilde{\mathbf{H}}_m$. Applying the Givens rotations to both the right- and left-hand-sides yields a modified matrix

$$\mathbf{R}_m = \begin{bmatrix} 1.0000 & 0.3341 & -0.0460 \\ 0.0000 & 1.0000 & -0.6305 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

and a modified right-hand-side vector

$$\mathbf{z} = \begin{bmatrix} 0.1442 \\ 0.0082 \\ -1.8569 \end{bmatrix}.$$

Here we have omitted the last row of both the matrix and right-hand-side vector because all the entries are zero as expected. We now solve the system $\mathbf{R}_m\, \mathbf{y} = \mathbf{z}$ using back substitution. We obtain the vector

$$\mathbf{y} = \begin{bmatrix} 0.4472 \\ -1.1626 \\ -1.8569 \end{bmatrix},$$

which provide us with the coefficients for the linear combination of the vectors of $\mathbf{V}_{m+1}$. Taking the linear combination of the first three columns of $\mathbf{V}_{m+1}$ (since $m = 3$), we obtain:

$$0.4472 \begin{bmatrix} 0.0000 \\ 0.4472 \\ 0.8944 \end{bmatrix} - 1.1626 \begin{bmatrix} 0.8305 \\ 0.4983 \\ -0.2491 \end{bmatrix} - 1.8569 \begin{bmatrix} 0.5571 \\ -0.7428 \\ 0.3714 \end{bmatrix} = \begin{bmatrix} -2.000 \\ 1.000 \\ 0.000 \end{bmatrix},$$

which is the exact solution.

9.5.3 GMRES(k)

The problem with the GMRES is that it requires storing all the vectors $\mathbf{v}_m$, which can become very expensive. To save storage and also computational cost we can run the GMRES process only for k steps and subsequently *restart it* with the vector $\mathbf{x}_k$ as an initial guess. This is the so-called *GMRES(k)* version of the generalized minimum residual method. The choice of k is crucial, as a value too small may lead to divergence while a large value results in extra computations.

Software Suite *Putting it into Practice*

Below we present a serial implementation of GMRES(m). This function takes as input the integer m denoting the size of the Krylov subspace to be used, the matrix $\mathbf{A}$, the right-hand-side vector $\mathbf{b}$, and the result vector $\mathbf{x}$. We automatically set the initial direction to $(1, 0, \ldots, 0)^T$. Two constants can be found in the function: *maxit* which specifies the maximum number of iterations before terminating and *tol* which specifies the stopping tolerance.

```
void GMRES(int m, const SCMatrix &A, const SCVector &b,
           SCVector &x){
  int i,j,k,ll,nr;
  int N = A.Rows();
  SCMatrix H(m+1,m),V(N,m+1);
  SCVector w(N),r(N),y(m+1),z(N);
  double * c = new double[m+1];
  double * s = new double[m+1];
  const int maxit = 1000;
  const double tol = 1.0e-7;
  double delta,rho,tmp;

  x.Initialize(0.0);

  r = b - A*x;

  for(j=0;j<maxit;j++){
    y.Initialize(0.0);
    y(0) = r.Norm_l2();
```

```
r.Normalize();

ModifiedArnoldi(m,r,A,H,V);

/* Givens Rotation to accomplish QR factorization */
for(i=0;i<m;i++){
  for(k=1;k<=i;k++){
    tmp = H(k-1,i);
    H(k-1,i) = c[k-1]*H(k-1,i) + s[k-1]*H(k,i);
    H(k,i) = -s[k-1]*tmp + c[k-1]*H(k,i);
  }

  delta = sqrt(H(i,i)*H(i,i)+H(i+1,i)*H(i+1,i));
  c[i] = H(i,i)/delta;
  s[i] = H(i+1,i)/delta;

  H(i,i) = c[i]*H(i,i) + s[i]*H(i+1,i);

  for(k=i+1;k<m+1;k++)
    H(k,i) = 0.0;

  y(i+1) = -s[i]*y(i);
  y(i)   =  c[i]*y(i);
  rho = fabs(y(i+1));
  if(rho < tol){
    nr = i;
    break;
  }
}

/* Backsolve to obtain coefficients */
z.Initialize(0.0);
if(i>=(m-1)){
  nr = m;
  z(nr-1) = y(nr-1)/H(nr-1,nr-1);
}

for(k=nr-2;k>=0;k--){
  z(k) = y(k);
  for(ll=k+1;ll<nr;ll++)
```

```
        z(k) -= H(k,ll)*z(ll);
      z(k) = z(k)/H(k,k);
    }

    /* Linear combination of basis vectors
       of the Krylov space                  */
    for(i=0;i<nr;i++){
      V.GetColumn(i,r);
      x = x + z(i)*r;
    }

    if(rho<tol)
      break;

    r = b - A*x;
  }
  delete[] c;
  delete[] s;
}
```

Remark 1: Observe that this function requires the storage of both **H** and **V**, both of which may be of the same size as the original matrix **A**. This fact is one of the primary motivations for introducing the restart parameter so that smaller Krylov spaces (and hence less storage) can be used.

Remark 2: Instead of forming the Givens matrices explicitly as we did in the example above, we can take advantage of the structure of the matrices so that we need not store the rotation matrices. Instead, we can loop through the appropriate positions, updating as we go.

9.5.4 Preconditioning GMRES

GMRES is used in practice when the matrix **A** is not well-conditioned. This means that convergence is typically slow and appropriate preconditioners should be employed. Similar to preconditioning of symmetric matrices where incomplete Cholesky was found to be effective, here we use *incomplete LU (ILU)*. Specifically, we employ the preconditioner

$$\mathbf{M} = \mathbf{L}\mathbf{U}$$

where **L** and **U** are the lower and upper triangular matrices corresponding to **A** but with no fills-in at the entries $a_{ij} = 0$. Note that the preconditioner should not be constructed explicitly but rather be incorporated in the Arnoldi iteration process. To this end, we need to insert the following code

```
⋮
$\mathbf{My} = \mathbf{v}_j$
$\mathbf{w} = \mathbf{Ay}$
for $i = 1, \dots, j$
    $h_{ij} = (\mathbf{w}, \mathbf{v}_i)$
    $\mathbf{w} = \mathbf{w} - h_{ij}\mathbf{v}_i$
⋮
```

in the modified Arnoldi iteration algorithm presented above.

Remark: GMRES employs long vectors to obtain orthogonality unlike the three-term recurrence formula associated with symmetric systems. For non-symmetric systems it is also possible to use three-term recurrence formula as done in the Biconjugate Gradient (BiCG) method, employing *two* mutually orthogonal sequences of vectors. A more stable version of BiCG is the Quasi-Minimal Residual (QMR) method which avoids possible break-downs and converges faster, i.e., as fast as GMRES [39]. Both BiCG and QMR solve tridiagonal systems corresponding to the three-term recurrence sequences. Details of implementation for both methods can be found in [5].

9.5.5 Parallel GMRES

One immediate complication when attempting to parallelize the serial algorithm previously presented is attempting to parallelize the modified Arnoldi component. The brute-force implementation leads to very poor scalability. Similar to the parallel PCG algorithm, we can also use blocked operations to increase the efficiency and parallelism. One way proposed in [64] is to first produce the following basis for the Krylov space

$$\mathbf{v}_1, \mathbf{A}\mathbf{v}_1, \dots, \mathbf{A}^k\mathbf{v}_1$$

and subsequently to orthogonalize the entire set. In contrast, in the standard GMRES method each new vector is immediately orthogonalized to all previous vectors. This approach increases significantly *data locality.*

Another approach is to employ BLAS2 routines as much as possible in GMRES instead of the obvious BLAS1, which is the least efficient. An algorithm proposed in [79] is to replace the modified Gram-Schmidt with the standard Gram-Schmidt but apply it twice. The double orthogonalization has been shown to reduce the numerical instability associated with the classical Gram-Schmidt method. In the context of more efficient computation, we can now compute all the *dot products* in parallel.

9.6 What Solver to Choose?

The question of what solver and what preconditioner to choose is a complex one, and in many cases there are more than one good candidates. From the algorithmic point of view, we have to consider the properties of the matrix $\mathbf{A}$, and examine if :

- $\mathbf{A}$ is symmetric or non-symmetric.
- $\mathbf{A}$ is positive-definite.
- Both $\mathbf{A}$ and $\mathbf{A}^T$ are available.
- $\mathbf{A}$ is sparse.
- $\mathbf{A}$ is ill-conditioned or not.
- Good preconditioner exists.

In addition, we have to consider the computational requirements and resources, namely:

- Parallel or serial computation.
- Vector or scalar processor.
- Multi-threading.
- Re-use of data in cache.
- Indirect addressing.
- Memory size.

Method	**ddot**	**daxpy**	**dgemv**	**Storage**
Jacobi			1	$3n$
SOR		1	1	$2n$
CGM	2	3	1	$6n$
GMRES	$j+1$	$j+1$	1	$(j+5)n$
QMR	2	12	2	$16n$

Table 9.2: Main operations and storage for iterative solvers; n is the matrix order and j denotes the iteration number. The storage shown does not include the matrix storage.

The above lists are indicative but not exhaustive of the issues that need to be considered in the decision regarding the choice of solver. What is obviously a faster code for serial computations is not necessarily faster on a parallel computer. In table 9.2 we list the main operations in terms of BLAS routines of the iterative solvers we studied in this book. Algorithms that employ *dgemv*, i.e., the matrix-vector multiply, are typically more efficient as this operation can be done efficiently both in a serial and in a parallel environment.

For problems involving differential equations, the type of differential equation we have to deal with and the corresponding discretization we choose defines the linear system we solve. For Poisson and Helmholtz equations we obtain symmetric systems. A typical order-of-magnitude cost in computational work and storage for various direct and iterative solvers is shown in table 9.3. The lower bound is of the same order of magnitude as the multigrid method. This clear advantage of multigrid, however, can easily be lost on a parallel computer as very sparse systems that need to be solved at the coarsest level of multigrid are not easily parallelizable. Hence, multigrid is not necessarily the fastest and Jacobi is not the slowest as the estimate for the serial work suggests. Similarly, the most effective (in terms of its spectrum and serial cost) preconditioner may not be the best overall preconditioner. Typically, the more sophisticated preconditioners are more complex and *simplicity* is the rule in parallel computing.

For large size problems the associated memory considerations suggest the use of iterative solvers. A possible decision tree for this case is as follows, see figure 9.17:

- If the matrix is symmetric and positive-definite then preconditioned

Method	**Direct/Iterative**	**Work**	**Storage**
Multigrid	I	n	n
SSOR/Chebyshev	I	$n^{5/4}$	n
SOR	I	$n^{3/2}$	n
CGM	I	$n^{3/2}$	n
Gauss-Seidel	I	n^2	n
Jacobi	I	n^2	n
Gauss Elimination/sparse	D	$n^{3/2}$	$n \cdot \log n$
Gauss Elimination/dense	D	n^3	n^2

Table 9.3: Computational work and storage for solution of a Poisson equation on a N-by-N finite difference grid; $n = N^2$. I stands for iterative and D for direct solver.

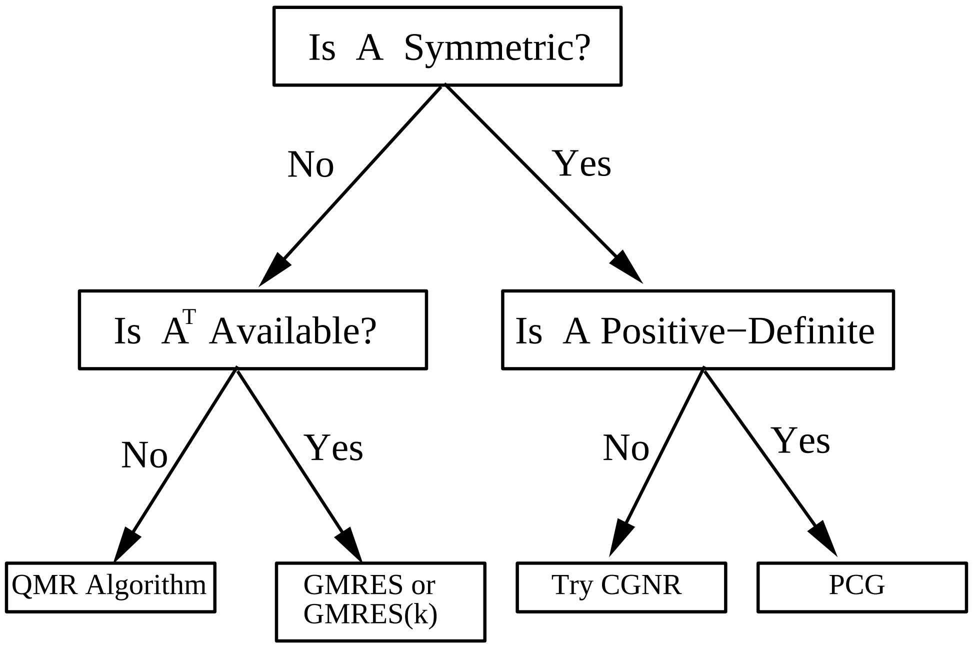

Figure 9.17: Decision tree for which algorithm to use based upon the properties of **A**.

conjugate gradient may be the best candidate. The question then becomes which preconditioner is the best – that is problem-dependent and computer-dependent.

- If **A** is not positive-definite use CGNR assuming the condition number is reasonable.
- On the other hand, if **A** is not symmetric then the first choice should be GMRES or GMRES(k) if memory is at a premium.
- However, if $\mathbf{A}^T$ is not available then the QMR algorithm which is quite robust and as fast as GMRES would be a good candidate – it works effectively even for ill-conditioned matrices.

This is just one of the many possible scenarios – what makes this field interesting is that the choices are not unique!

9.7 Available Software for Fast Solvers

The basic algorithms we have presented in this chapter are relatively easy to program but more sophisticated versions with respect to preconditioning, restarts, orthogonalization, and parallel implementation are available in free software at:

- www.netlib.org

The direct solvers are part of ScaLAPACK and LAPACK++ while the iterative solvers are part of the Templates package [5].

Specifically in ScaLAPACK/LAPACK++, for LU type operations the routine **SGETRF** performs LU factorization with pivoting based on the BLAS3 routines and therefore it is very efficient. A similar routine **SGETF2** is based on BLAS2 routines, and it is also efficient. For the specific implementations involved see [26]. For symmetric positive-definite matrices the routines **SPOTRF** and **SPOTRS** perform Cholesky factorization of a matrix and solve a linear system, respectively. Finally, the routine **SPTTRF** performs an LDL^T factorization of a symmetric positive-definite matrix.

Also in ScaLAPACK/LAPACK++, for QR type operations the routine **SGEQRF** performs QR factorization and the routine **SGEQPF** performs QR factorization with column pivoting. The routine **SGERQF** performs RQ factorization while the routine **SGEHRD** reduces a general matrix to upper Hessenberg form.

The **cpptemplates** files available at www.netlib.org contain implementations of all the iterative solvers presented in this chapter for matrix-vector classes. In particular, the routines declared in **cg.h** and **cheby.h** implement the conjugate gradient method and the preconditioned Chebyshev method, respectively, for symmetric positive-definite systems. The routines declared in **cgs.h**, **gmres.h**, **qmr.h** and **bicg.h** are suitable for non-symmetric systems and their names indicate the corresponding algorithms.

9.8 Homework Problems

1. Find the condition number of the matrices

$$A = \begin{bmatrix} 0.001 & 1 \\ 1 & 1 \end{bmatrix} \text{ and } B = \begin{bmatrix} 7 & 6.990 \\ 4 & 4 \end{bmatrix}$$

2. Let $\mathbf{A} = \mathbf{L}\ \mathbf{D}\mathbf{L}^T$ be a symmetric positive-definite matrix, and $\mathbf{D} =$ diag (d_{ii}). Then, show that

$$\kappa_2(\mathbf{A}) \geq \frac{\max(d_{ii})}{\min(d_{ii})},$$

where $\kappa_2(\mathbf{A})$ is the condition number of $\mathbf{A}$ in the L_2-norm.

3. For what values of ϵ the matrix

$$\begin{bmatrix} 1 & \epsilon \\ \epsilon & 1 \end{bmatrix}$$

is ill-conditioned? How will you results be affected if you are to compute in *single* or *double* precision?

4. Let us assume that the matrix $\mathbf{A}$ is *strictly* diagonally dominant, i.e.,

$$\sum_{i \neq j} |a_{ij}| < |a_{ii}| \, .$$

Show that if you apply the LU factorization procedure to $\mathbf{A}$ with partial pivoting, it has no effect on the rows, that is no actual row exchange occurs. This proves what we discussed in section 9.1.2 that no pivoting is required for a strictly diagonally-dominant matrix.

5. (*Almost triangular matrix - Hessenberg*)

   This matrix is defined by

$$\begin{aligned} a_{ij} &= 0, \quad i > j + 1 \\ a_{ij} &\neq 0, \quad i \leq j + 1 \end{aligned}$$

   Estimate the operation count for the LU factorization and the backward solve for this matrix.

6. Consider the matrix $\mathbf{A}$ written in a block 2×2 form with submatrices $\mathbf{A}_{ij}$, $i, j = 1, 2$ of equal size $m \times m$. Show that the Schur complement defined as
$$\mathbf{S} \equiv \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$$
overwrites the matrix $\mathbf{A}_{22}$ after m steps of Gaussian elimination without pivoting.

7. Apply Hager's algorithm (by-hand calculation) to a (3×3) matrix whose rows ($\mathbf{r}_i$, $i = 1, 2, 3$) are: $\mathbf{r}_i = i + j - 1$, $j = 1, 2, 3$. Compute also the exact condition number in the L_1-norm and compare the two values.

8. Apply Hager's algorithm to estimate the condition number of the Hilbert $(n \times n)$ matrix defined by
$$h_{ij} = \frac{1}{i + j - 2}$$
for n = 4, 16, 32. (Use the LAPACK routines or any other available routines). What do you observe ?

9. In the parallel LU program, we did not implement row pivoting. Modify the code given in the text to accomplish row pivoting.

10. Using the parallel LU program as a guide, implement an MPI program to accomplish Cholesky Factorization.

11. Apply the *incomplete* Cholesky factorization to the tridiagonal matrix (-1, 4, -1). Compare the results with the results from the standard Cholesky factorization. Does your conclusion hold for any banded matrix?

12. In the cyclic reduction code presented in this chapter, the matrix $\mathbf{A}$ is allocated as if it were a full matrix. Modify both the serial and the parallel code so that only the necessary amount of memory is used to store the tridiagonal matrix $\mathbf{A}$.

13. Consider the following system:
$$\begin{aligned} -2x_1 + x_2 &= F_1 \\ x_1 - 2x_2 + 2x_3 &= F_2 \\ x_2 - 2x_3 + 2x_4 &= F_3 \\ x_3 - 2x_4 + 2x_5 &= F_4 \end{aligned}$$

$$\begin{aligned} x_4 - 2x_5 + 2x_6 &= F_5 \\ x_5 - 2x_6 + 2x_7 &= F_6 \\ x_6 - 2x_7 &= F_7 \end{aligned}$$

Solve this system by hand using cyclic reduction. At what level does this system terminate, and with what equation? Is there sufficient information at the point of termination to obtain the solution? If so, accomplish the back substitution to obtain the answer.

14. Use the Householder transformation to show that if $\mathbf{H}_e$ is a Hessenberg matrix and $\mathbf{H}_e = \mathbf{QR}$ then the matrix $\mathbf{H}_e^* = \mathbf{RQ}$ is also a Hessenberg matrix.

15. The flop count for solving the overdetermined system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is of size $m \times n$ is (choose one)

    (a) $\dfrac{mn^2}{2} + \dfrac{n^3}{6}$ for normal equations and $\dfrac{mn^2}{2} - \dfrac{n^3}{6}$ for the Householder QR method.

    (b) The reverse of (a).

    (c) None of the above.

16. The flop count for QR factorization of an $n \times n$ matrix with column pivoting using the Householder method is (choose one)

    (a) $\dfrac{5}{3}n^3$.

    (b) $\dfrac{5}{4}n^3$.

    (c) $\dfrac{4}{5}n^2$.

17. The number of additions and multiplications in Cholesky factorization is roughly half that of LU factorization.

    (a) True.

    (b) False.

18. The QR method conserves the bandwidth of a matrix (choose one)

    (a) Always true.

(b) In some cases.

(c) Never.

19. The QR factorization of a matrix **A** is (choose one)

(a) Always unique.

(b) Unique if **A** is non-singular.

(c) Not unique.

20. The most efficient way of distributing a matrix on a parallel machine is by (choose one)

(a) Rows.

(b) Columns.

(c) Depends on the problem one is trying to solve.

21. Define $h = \frac{1}{N}$, $\mu = h^2$, and $q[i][j] = (8\pi^2 + 1)h^2 sin(2\pi hi) sin(2\pi hj)$ where $i, j = 0, \ldots, N-1$. Let **A** be of the form given in figure 9.18.

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
4+\mu & -1 & & -1 & -1 & & & & & & & & -1 & & & \\
-1 & 4+\mu & -1 & & & -1 & & & & & & & & -1 & & \\
 & -1 & 4+\mu & -1 & & & -1 & & & & & & & & -1 & \\
-1 & & -1 & 4+\mu & & & & -1 & & & & & & & & -1 \\
\hline
-1 & & & & 4+\mu & -1 & & -1 & -1 & & & & & & & \\
 & -1 & & & -1 & 4+\mu & -1 & & & -1 & & & & & & \\
 & & -1 & & & -1 & 4+\mu & -1 & & & -1 & & & & & \\
 & & & -1 & -1 & & -1 & 4+\mu & & & & -1 & & & & \\
\hline
 & & & & -1 & & & & 4+\mu & -1 & & -1 & -1 & & & \\
 & & & & & -1 & & & -1 & 4+\mu & -1 & & & -1 & & \\
 & & & & & & -1 & & & -1 & 4+\mu & -1 & & & -1 & \\
 & & & & & & & -1 & -1 & & -1 & 4+\mu & & & & -1 \\
\hline
-1 & & & & & & & & -1 & & & & 4+\mu & -1 & & -1 \\
 & -1 & & & & & & & & -1 & & & -1 & 4+\mu & -1 & \\
 & & -1 & & & & & & & & -1 & & & -1 & 4+\mu & -1 \\
 & & & -1 & & & & & & & & -1 & -1 & & -1 & 4+\mu
\end{array}\right]
*
\begin{bmatrix} u[0][0] \\ u[1][0] \\ u[2][0] \\ u[3][0] \\ u[0][1] \\ u[1][1] \\ u[2][1] \\ u[3][1] \\ u[0][2] \\ u[1][2] \\ u[2][2] \\ u[3][2] \\ u[0][3] \\ u[1][3] \\ u[2][3] \\ u[3][3] \end{bmatrix}
=
\begin{bmatrix} q[0][0] \\ q[1][0] \\ q[2][0] \\ q[3][0] \\ q[0][1] \\ q[1][1] \\ q[2][1] \\ q[3][1] \\ q[0][2] \\ q[1][2] \\ q[2][2] \\ q[3][2] \\ q[0][3] \\ q[1][3] \\ q[2][3] \\ q[3][3] \end{bmatrix}$$

Figure 9.18: Matrix system $\mathbf{Au} = \mathbf{q}$ for $N = 4$.

Solve the matrix system $\mathbf{Au} = \mathbf{q}$ for **u** using the following for methods:

(a) Serial Jacobi.

(b) Parallel Jacobi.

(c) Serial Conjugate Gradient.

(d) Serial Preconditioned Conjugate Gradient, preconditioned using incomplete Cholesky.

(e) Parallel Conjugate Gradient.

(f) Parallel Preconditioned Conjugate Gradient, preconditioned using incomplete Cholesky.

Solve for both $N = 4$ and $N = 20$. Here N denotes the number of points used in both the x and y directions (hence the total number of grid points is N^2 points, which corresponds to the rank of the matrix for which we are solving). For the serial algorithms, provide a graphical plot of the solution $u[i][j]$ at the points (hi, hj) (either a contour or surface plot). For the parallel algorithms, show the parallel speed-up using different number of processors (this will require you to use the *MPI_Wtime* function to time your runs).

22. Solve the Poisson equation on a three-dimensional grid $N \times N \times N$ using a second-order finite difference discretization with Dirichlet boundary conditions. Employ the following (serial) algorithms:

    (a) Conjugate Gradients.

    (b) Conjugate Gradients with incomplete Cholesky as preconditioner.

    (c) SSOR with Chebyshev acceleration.

    (d) Jacobi.

    Estimate the computational work in terms of $\mathcal{O}(n^\alpha)$, that is find α. Verify these estimates by solving for $n = N^3 = 64^3$ and timing your codes on the same computer.

23. In the text we presented a QR factorization example using the Hilbert matrix. At the conclusion of the example, we had constructed the matrix $\mathbf{R}$. From the values of $\mathbf{w}$ found in the example, compute the matrix $\mathbf{Q}$ and show that indeed $\mathbf{A} = \mathbf{QR}$.

    Hint: Recall that the Householder matrix is given by

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{w}\mathbf{w}^T}{\mathbf{w}^T\mathbf{w}}.$$

    Construct $\mathbf{H}_1$ and $\mathbf{H}_2$ and use them to construct $\mathbf{Q}$ by the expressions given in the text.

24. Modify the QR routine in the text so that it computes the value of $\mathbf{Q}$. You will need to modify the input arguments of the function to accept a SCMatrix Q which you should fill in with the appropriate values.

25. Write a serial function which accomplishes QR decomposition of a tridiagonal matrix using Givens rotations. The function should have at least the following three arguments: the matrix $\mathbf{A}$ as input and the matrices $\mathbf{Q}$ and $\mathbf{R}$ as output.

26. Create a function which solves the system $\mathbf{Ax} = \mathbf{b}$ for tridiagonal matrices $\mathbf{A}$, and uses the QR code you wrote previously. What property of both $\mathbf{Q}$ and $\mathbf{R}$ can we use to accomplish this efficiently?

27. Write a parallel program which accomplishes QR decomposition of a tridiagonal matrix using Givens rotations. Partition the matrix $\mathbf{A}$ by rows across the processors. Design your program so that each processor has the rows of $\mathbf{Q}$ and $\mathbf{R}$ which correspond to the rows of $\mathbf{A}$ which reside on the processor.

28. Consider five symmetric Toeplitz matrices $\mathbf{A}$ with entries given by ($k = 1, \ldots, n$)

$$a_k^{(1)} = 1/k; \quad a_k^{(2)} = 1/\sqrt{k}; \quad a_k^{(3)} = 1/k^2; \quad a_k^{(4)} = k; \quad a_k^{(5)} = \cos k/k\,,$$

and an arbitrary non-zero vector $\mathbf{b}$. Use PCGM to solve the systems $\mathbf{Ax} = \mathbf{b}$ with circulant preconditioners and experiment with different types (i.e., number of Toeplitz diagonals employed). Compare your results without CG preconditioning in terms of the number of iterations for sizes up to $n = 100$ and tolerance levels just above single machine accuracy.

29. Estimate the operation count for GMRES(k) for fixed k and compare it with GMRES assuming a large value of the order n of matrix $\mathbf{A}$. Does GMRES(k) converge for any value of k?

30. For the GMRES example problem given in the text, use the function provided to attempt GMRES(2). Add *cout* statements to keep report the residual. Try a variety of different initial guesses $\mathbf{x}_0$ and plot the residual versus iteration for each case. Is the convergence rate the same?

31. Consider the linear advection-diffusion equation

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}$$

where ν is the diffusion coefficient, in the domain

$$x \in [0, 10] \quad \text{and} \quad t \in [0, 5].$$

We also assume periodic boundary conditions and that the initial conditions are:

$$u(0 \leq x \leq 1; 0) = x(1-x) \quad \text{and} \quad u(1 \leq x \leq 10; 0) = 0 \quad .$$

Employ a second-order upwind for the advection and a central finite difference scheme for the diffusion (with a Crank-Nicolson in time) to discretize this problem. Invert the resulted system using GMRES(k) and experiment with different values of k to minimize the solution time. Use $n = 128$ points for the discretization. Does the value of optimum k depends on the time step Δt. Is this method unconditionally stable?