```
void ThomasAlgorithm_per(int N, double *b, double *a, double *c,
                         double *x, double *q){
  int i;
  double *x1,*x2,*q2;

  x1 = new double[N-1];
  x2 = new double[N-1];
  q2 = new double[N-1];

  /* Prepare secondary q */
  for(i=0;i<N-1;i++)
    q2[i] = 0.0;
  q2[0] = -b[N-1];
  q2[N-2] = -c[N-2];

  ThomasAlgorithm(N-1,b,a,c,x1,q);
  ThomasAlgorithm(N-1,b,a,c,x2,q2);

  x[N-1] = (q[N-1] - c[N-1]*x1[0] - b[N-2]*x1[N-2])/
    (a[N-1] + c[N-1]*x2[0] + b[N-2]*x2[N-2]);

  for(i=0;i<N-1;i++)
    x[i] = x1[i] + x2[i]*x[N-1];

  delete[] x1;
  delete[] x2;
  delete[] q2;
}
```

Key Concept

- Code re-use is important. If you have already invested the time to make sure that a routine is well-written and correctly implemented, then you can use the routine as a component in new routines.

6.1.5 Parallel Algorithm for Tridiagonal Systems

In seeking a parallelization strategy for solving triagonal systems, we will once again examine the structure of the LU decomposition as we did in formulating the Thomas algorithm. By exploiting the recursive nature of the LU decomposition, we will devise a *full-recursive-doubling* procedure for solving for the unknown LU coefficients. For a more detailed description of the algorithm which follows we refer the reader to [24].

As before, we seek an LU decomposition of the tridiagonal matrix $\mathbf{A}$ as follows:

$$\left[\underbrace{\begin{matrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & 0 & \\ & b_3 & a_3 & c_3 & \\ & \ddots & \ddots & \ddots & \\ & 0 & & b_N & a_N \end{matrix}}_{\mathbf{A}}\right] = \left[\underbrace{\begin{matrix} 1 & & & & \\ \ell_2 & 1 & & 0 & \\ & \ell_3 & 1 & & \\ & & \ddots & \ddots & \\ & 0 & & \ell_N & 1 \end{matrix}}_{\mathbf{L}}\right]\left[\underbrace{\begin{matrix} d_1 & u_1 & & & \\ & d_2 & u_2 & 0 & \\ & & d_3 & u_3 & \\ & & 0 & \ddots & \ddots \\ & & & & d_N \end{matrix}}_{\mathbf{U}}\right].$$

Upon examination of the expression above, we see that we can formulate recurrence relations for the unknown coefficients d_j, u_j, and l_j as follows:

$$\begin{aligned} a_1 &= d_1 && (6.5)\\ c_j &= u_j && (6.6)\\ a_k &= d_k + l_k u_{k-1} && (6.7)\\ b_k &= l_k d_{k-1} && (6.8) \end{aligned}$$

where $j = 1, \dots, N$ and $k = 2, \dots, N$. Given equation (6.6), we can immediately solve for all the unknown coefficients u_j. To solve for d_j and l_j, we rely on the recursive nature of these equations. Substituting equations (6.6) and (6.8) into equation (6.7) and rearranging terms yields the following rational recursion relationship for the unknown coefficient d_j:

$$\begin{aligned} d_j &= a_j - l_j u_{j-1} \\ &= a_j - \frac{b_j}{d_{j-1}} u_{j-1} \\ &= \frac{a_j d_{j-1} - b_j c_{j-1}}{d_{j-1} + 0}. \end{aligned}$$

We can then inductively solve for the all coefficients d_j, and use this information along with equation 6.8 to solve for l_j.

To parallelize this procedure, we make use of a full-recursive-doubling procedure on the sequence of 2×2 matrices given by:

$$\mathbf{R}_0 = \begin{bmatrix} a_0 & 0 \\ 1 & 0 \end{bmatrix}$$

and

$$\mathbf{R}_j = \begin{bmatrix} a_j & -b_j c_{j-1} \\ 1 & 0 \end{bmatrix}$$

for $j = 1, \ldots, N$. Using the Möbius transformations

$$\mathbf{T}_j = \mathbf{R}_j \mathbf{R}_{j-1} \ldots \mathbf{R}_0$$

we have that

$$d_j = \frac{\begin{pmatrix} 1 \\ 0 \end{pmatrix}^t \mathbf{T}_j \begin{pmatrix} 1 \\ 1 \end{pmatrix}}{\begin{pmatrix} 0 \\ 1 \end{pmatrix}^t \mathbf{T}_j \begin{pmatrix} 1 \\ 1 \end{pmatrix}}.$$

To explain how this information can be used for parallelization, we will examine a specific example. Suppose that we are given a tridiagonal matrix $\mathbf{A}$ of size 40, and that we want to solve the problem using 8 processes. Assume that all processes have a copy of the original matrix $\mathbf{A}$. We first partition the matrix such that each process is responsible for five rows: process P_0 is responsible for rows 0-4, P_1 is responsible for rows 5-9, etc. We then accomplish the following steps:

1. On each process P_j form the matrices $\mathbf{R}_k$, where k corresponds to the row indices for which the process is responsible, and ranges between k_min and k_max.
2. On each process P_j form the matrix $\mathbf{S}_j = \mathbf{R}_{k\_max}\mathbf{R}_{k\_max-1} \ldots \mathbf{R}_{k\_min}$.
3. Using the full-recursive-doubling communication pattern as given in table 6.1, distribute and combine the $\mathbf{S}_j$ matrices as given in table 6.2.
4. On each process P_j calculate the local unknown coefficients d_k $(k\_min \leq k \leq k\_max)$ using local R_k and matrices obtained from the full-recursive-doubling.
5. For processes P_0 through P_6, send the local d_{k_max} to the process one process *id* up (i.e., P_0 sends to P_1; P_1 sends to P_2; etc.).
6. On each process P_j calculate the local unknown coefficients l_k $(k\_min \leq k \leq k\_max)$ using the local d_k values and the value obtained in the previous step.
7. Distribute the d_j and l_j values across all processes so that each process has all the d_j and l_j coefficients.
8. On each process P_j perform a local forward and backward substitution to obtain the solution.

Software Suite *Putting it into Practice*

Stage 1	Stage 2	Stage 3
$P_0 \to P_1$	$P_0 \to P_2$	$P_0 \to P_4$
$P_1 \to P_2$	$P_1 \to P_3$	$P_1 \to P_5$
$P_2 \to P_3$	$P_2 \to P_4$	$P_2 \to P_6$
$P_3 \to P_4$	$P_3 \to P_5$	$P_3 \to P_7$
$P_4 \to P_5$	$P_4 \to P_6$	
$P_5 \to P_6$	$P_5 \to P_7$	
$P_6 \to P_7$		

Table 6.1: Full-recursive-doubling communication pattern. The number of stages is equal to the $\log_2 M$ where M is the number of processes. In this case, $M = 8$ and hence there are three stages of communication.

Process	Stage 0	Stage 1	Stage 2	Stage 3
P_0	S_0			
P_1	S_1	S_1S_0		
P_2	S_2	S_2S_1	$S_2S_1S_0$	
P_3	S_3	S_3S_2	$S_3S_2S_1S_0$	
P_4	S_4	S_4S_3	$S_4S_3S_2S_1$	$S_4S_3S_2S_1S_0$
P_5	S_5	S_5S_4	$S_5S_4S_3S_2$	$S_5S_4S_3S_2S_1S_0$
P_6	S_6	S_6S_5	$S_6S_5S_4S_3$	$S_6S_5S_4S_3S_2S_1S_0$
P_7	S_7	S_7S_6	$S_7S_6S_6S_4$	$S_7S_6S_5S_4S_3S_2S_1S_0$

Table 6.2: Distribution and combination pattern of the S_j matrices for each stage. The interpretation of the table is as follows: Given the communication pattern as given in table 6.1, in stage one P_0 sends S_0 to P_1, which P_1 combines with its local S_1 to form the product S_1S_0. Similarly in stage one, P_1 sends S_1 to P_2, etc. In stage two, P_0 sends S_0 to P_2, which P_2 combines with its local product S_2S_1 to form $S_2S_1S_0$. Similarly P_1 sends S_1S_0 to P_3 which is then combined on P_3 to form $S_3S_2S_1S_0$. In stage three, the final communications occur such that each process j stores locally the product $S_jS_{j-1}\ldots S_0$.

We now present a *parallel Thomas algorithm function* which uses the full-recursive-procedure discussed above. This function assumes that the MPI initialization has already been accomplished by the calling function, and it requires that the number of processes used is a *power of two*. It takes as input its process *id number*, the total number of processes being used, the size of the matrix, the matrix **A** stored in the arrays a, b, and c as before, and the right-hand-side vector **q** stored in the array q. The output of this function on all processes is the solution vector contained within the array x. We first present the function definition and then present some remarks on the code.

```
void ThomasAlgorithm_P(int mynode, int numnodes, int N, double *b,
                       double *a, double *c, double *x, double *q){
  int i,j,k,i_global;
  int rows_local,local_offset;
  double S[2][2],T[2][2],s1tmp,s2tmp;
  double *l,*d,*y;
  MPI_Status status;

  l = new double[N];
  d = new double[N];
  y = new double[N];

  for(i=0;i<N;i++)
    l[i] = d[i] = y[i] = 0.0;

  S[0][0] = S[1][1] = 1.0;
  S[1][0] = S[0][1] = 0.0;

  rows_local = (int) floor(N/numnodes);
  local_offset = mynode*rows_local;

  // Form local products of R_k matrices
  if(mynode==0){
    s1tmp = a[local_offset]*S[0][0];
    S[1][0] = S[0][0];
    S[1][1] = S[0][1];
    S[0][1] = a[local_offset]*S[0][1];
    S[0][0] = s1tmp;
    for(i=1;i<rows_local;i++){
      s1tmp = a[i+local_offset]*S[0][0] -
              b[i+local_offset-1]*c[i+local_offset-1]*S[1][0];
      s2tmp = a[i+local_offset]*S[0][1] -
              b[i+local_offset-1]*c[i+local_offset-1]*S[1][1];
      S[1][0] = S[0][0];
      S[1][1] = S[0][1];
      S[0][0] = s1tmp;
      S[0][1] = s2tmp;
    }
  }
  else{
    for(i=0;i<rows_local;i++){
      s1tmp = a[i+local_offset]*S[0][0] -
              b[i+local_offset-1]*c[i+local_offset-1]*S[1][0];
      s2tmp = a[i+local_offset]*S[0][1] -
              b[i+local_offset-1]*c[i+local_offset-1]*S[1][1];
      S[1][0] = S[0][0];
      S[1][1] = S[0][1];
      S[0][0] = s1tmp;
      S[0][1] = s2tmp;
    }
  }

  // Full-recursive doubling algorithm for distribution
  for(i=0; i<=log2(numnodes);i++){
    if(mynode+pow(2,i) < numnodes)
      MPI_Send(S,4,MPI_DOUBLE,int(mynode+pow(2,i)),0,
               MPI_COMM_WORLD);
    if(mynode-pow(2,i)>=0){
      MPI_Recv(T,4,MPI_DOUBLE,int(mynode-pow(2,i)),0,
               MPI_COMM_WORLD,&status);
      s1tmp = S[0][0]*T[0][0] + S[0][1]*T[1][0];
      S[0][1] = S[0][0]*T[0][1] + S[0][1]*T[1][1];
      S[0][0] = s1tmp;
      s1tmp = S[1][0]*T[0][0] + S[1][1]*T[1][0];
      S[1][1] = S[1][0]*T[0][1] + S[1][1]*T[1][1];
      S[1][0] = s1tmp;
    }
  }

  //Calculate last d_k first so that it can be distributed,
  //and then do the distribution.
  d[local_offset+rows_local-1] = (S[0][0] + S[0][1])/
                                 (S[1][0] + S[1][1]);
  if(mynode == 0){
    MPI_Send(&d[local_offset+rows_local-1],1,MPI_DOUBLE,
             1,0,MPI_COMM_WORLD);
  }
  else{
    MPI_Recv(&d[local_offset-1],1,MPI_DOUBLE,mynode-1,0,
             MPI_COMM_WORLD,&status);
    if(mynode != numnodes-1)
      MPI_Send(&d[local_offset+rows_local-1],1,MPI_DOUBLE,
               mynode+1,0,MPI_COMM_WORLD);
  }


  // Compute in parallel the local values of d_k and l_k
  if(mynode == 0){
    l[0] = 0;
    d[0] = a[0];
    for(i=1;i<rows_local-1;i++){
      l[local_offset+i] = b[local_offset+i-1]/
                          d[local_offset+i-1];
      d[local_offset+i] = a[local_offset+i] -
                          l[local_offset+i]*c[local_offset+i-1];
    }
    l[local_offset+rows_local-1] = b[local_offset+rows_local-2]/
                                   d[local_offset+rows_local-2];
  }
  else{
    for(i=0;i<rows_local-1;i++){
      l[local_offset+i] = b[local_offset+i-1]/
                          d[local_offset+i-1];
      d[local_offset+i] = a[local_offset+i] -
                          l[local_offset+i]*c[local_offset+i-1];
    }
    l[local_offset+rows_local-1] = b[local_offset+rows_local-2]/
                                   d[local_offset+rows_local-2];
  }

  /*************************************************************/

  if(mynode>0)
    d[local_offset-1] = 0;

  // Distribute d_k and l_k to all processes

  double * tmp = new double[N];
  for(i=0;i<N;i++)
    tmp[i] = d[i];
  MPI_Allreduce(tmp,d,N,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  for(i=0;i<N;i++)
    tmp[i] = l[i];
  MPI_Allreduce(tmp,l,N,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  delete[] tmp;
```

```
  if(mynode ==0){
    /* Forward Substitution [L][y] = [q] */
    y[0] = q[0];
    for(i=1;i<N;i++)
      y[i] = q[i] - l[i]*y[i-1];

    /* Backward Substitution [U][x] = [y] */
    x[N-1] = y[N-1]/d[N-1];
    for(i=N-2;i>=0;i--)
      x[i] = (y[i] - c[i]*x[i+1])/d[i];

  }

  delete[] l;
  delete[] y;
  delete[] d;
  return;
}
```

Remark 1: Since we know that we are dealing with 2×2 matrices, we have chosen to allocate the 2×2 S array statically. It is important to note that when static allocation of arrays is used, the memory allocation is contiguous and in row-major order as shown in figure 6.13. We can use the contiguousness of the block of memory to our advantage when using MPI. Since S is stored as one contiguous block in memory, we can send the entire array in one MPI call instead of having to send the array row by row (as in the case where each row was dynamically allocated using the **new** command).

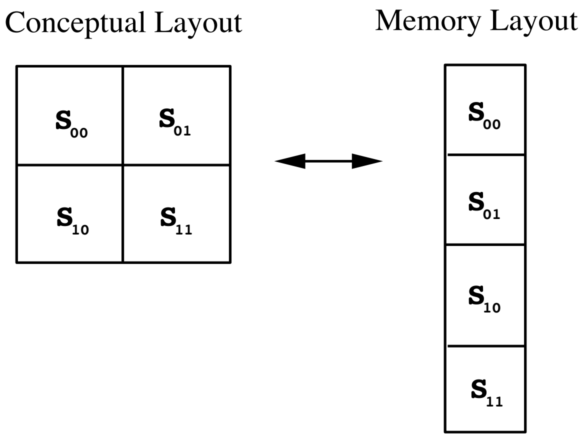

Figure 6.13: Memory layout of the matrix S. The double indexed array S is stored in a contiguous block of memory in row-major order.

Remark 2: Sometimes it becomes advantageous to use the reduction operator to mimic a gathering operation. We pictorially demonstrate how this can be accomplished in figure 6.14. In the code above, we use this trick to gather all the d_j and l_j values across all processors.

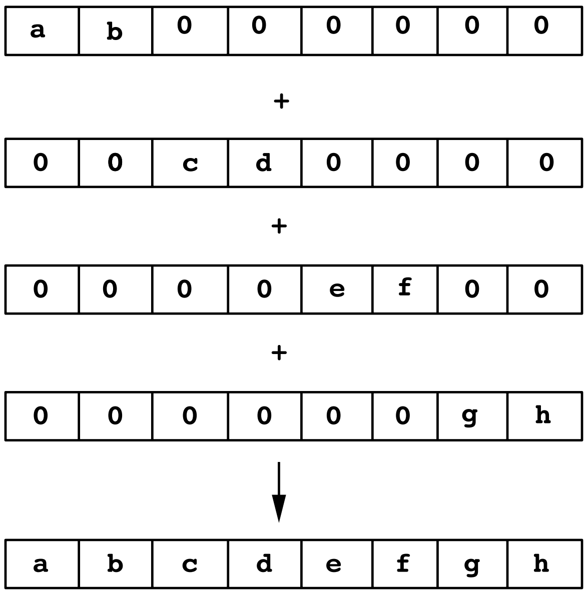

Figure 6.14: How to use the reduction operator to mimic the gathering process. In this example, we have four processes, each of which has two unique items to contribute. On each process, all other entries in the array are zeroed and then a sum is performed. The result is that the data is gathered into one array.

MPI Implementation Issues

In the sections above, we presented a serial and a parallel version of the Thomas algorithm. How can we time our parallel program to examine the speed-up due to adding more processes? MPI provides two functions which allow us to accomplish this task: *MPI_Wtime* and *MPI_Wtick* . We will now present for these two functions the function call syntax, argument list explanation, usage example and some remarks.

Function Call Syntax

double MPI_Wtime(void);

double MPI_Wtick(void);

Understanding the Argument Lists

- MPI_Wtime and MPI_Wtick take no arguments.

Example of Usage

```
int mynode, totalnodes;
double starttime, finaltime, precision;
```

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

precision = MPI_Wtick();
starttime = MPI_Wtime();

// Execution of commands here

finaltime = MPI_Wtime();

if(mynode == 0){
  cout << "The execution time was : " << finaltime-starttime;
  cout << " sec. with a precision of " << precision;
  cout << " sec." << endl;
}
```

Remarks

- These commands are very useful both for determining the parallel speed-up of your algorithm and for determining the components of your program which are using the most time.
- These commands provide to you the wallclock time (the physical time which has elapsed), not specifically the CPU time or communication time.

One question you may ask is how do I know that all the processes are exactly at the same point (assuming that I am doing the timing only on process 0)? MPI provides a function for synchronizing all processes called *MPI_Barrier*. When *MPI_Barrier* is called, the function will not return until all processes have called *MPI_Barrier*. This functionality allows you to synchronize all the processes, knowing that all processes exit the *MPI_Barrier* call at the same time. We will now present the function call syntax, argument list explanation, usage example and some remarks.

MPI_Barrier

Function Call Syntax

```
int MPI_Barrier(
        MPI_Comm   comm   /*  in  */,
```

Understanding the Argument Lists

- *comm* - communicator

Example of Usage

```
int mynode, totalnodes;

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

MPI_Barrier(MPI_COMM_WORLD);

// At this stage, all processes are synchronized
```

Remarks

- This command is a useful tool to help insure synchronization between processes. For example, you may want all processes to wait until one particular process has read in data from disk. Each process would call *MPI_Barrier* in the place in the program where the synchronization is required.