Description for Implementation of MPI Programs

SOLUTION OF SYSTEM OF LINEAR EQUATIONS USING JACOBI METHOD

BACKGROUND

The system of linear equations [A]{x} = {b} is given by

$$
\begin{aligned}
a_{o,o}x_o + a_{o,1}x_1 + ... + a_{o,n-1}x_{n-1} &= b_o ,\\
a_{1,o}x_o + a_{1,1}x_1 + ... + a_{1,n-1}x_{n-1} &= b_1 ,\\
&\cdots\\
&\cdots\\
&\cdots\\
a_{n-1,o}x_0 + a_{n-1,1}x_1 + ... + a_{n-1,n-1}x_{n-1} &= b_{n-1} .
\end{aligned}
$$

In matrix notation, this system of linear equation is written as $[\mathbf{A}]\{x\}=\{b\}$ where $\mathbf{A}[i,j] = a_{i,j}$, b is an n x 1 vector $[b_o,b_1,...,b_{n-1}]^T$, and is the desired solution vector $[x_o,x_1,...,x_{n-1}]^T$. We will make all subsequent references to $a_{i,j}$ by $\mathbf{A}[i,j]$ and x_i by $x[i]$. The matrix A is partitioned using *block row-wise* striping. Each process has n/p rows of the matrix A and the vector b. Since only process with rank 0 performs input and output, the matrix and vector must be distributed to the other processes.

Iterative methods are techniques to solve systems of equations of the form [A]{x}={b} that generate a sequence of approximations to the solution vector x. In each iteration, the coefficient matrix A is used to perform a matrix-vector multiplication. The number of iterations required to solve a system of equations with a desired precision is usually data dependent; hence, the number of iterations is not known prior to executing the algorithm. Therefore, we can analyze the performance and scalability of a single iteration of an iterative method. Iterative methods do not guarantee a solution for all systems of equations. However, when they do yield a solution, they are usually less expensive than direct methods for matrix factorization for large size of matrices.

CONJUGATE GRADIENT METHOD

BACKGROUND

Iterative methods are techniques to solve systems of equations of the form $[\mathbf{A}]\{x\}=\{b\}$ that generate a sequence of approximations to the solution vector x. In each iteration, the coefficient matrix **A** is used to perform a matrix-vector multiplication. The number of iterations required to solve a system of equations depends on input data matrix and hence, the number of iterations is not known prior to executing the algorithm. Iterative methods do not guarantee a solution for all systems of equations. However, when they do yield a solution, they are usually less expensive than direct methods for matrix factorization for large matrix system of equations.

SOLUTION OF SYSTEM OF LINEAR EQUATIONS USING GAUSSIAN ELIMINATION METHOD

BACKGROUND

The system of linear equations [A]{x} = {b} is given by

$a_{o,o}x_o + a_{o,1}x_1 + ... + a_{o,n-1}x_{n-1} = b_o$, $a_{1,o}x_o + a_{1,1}x_1 + ... + a_{1,n-1}x_{n-1} = b_1$,

....

..... $a_{n-1,o}x_0 + a_{n-1,1}x_1 + ... + a_{n-1,n-1}x_{n-1} = b_{n-1}$.

In matrix notation, this system of linear equation is written as $[\mathbf{A}]\{x\}=\{b\}$ where $\mathbf{A}[i, j] = a_{i, j}$, b is an n x 1 vector $[b_0,b_1,...,b_{n-1}]^T$, and is the desired solution vector $[x_0,x_1,...,x_{n-1}]^T$. We will make all subsequent references to $a_{i, j}$ by $\mathbf{A}[i, j]$ and x_i by $x[i]$. The matrix A is partitioned using *block row-wise* striping. Each process has *n/p* rows of the matrix A and the vector *b*. Since only process with rank *0* performs input and output, the matrix and vector must be distributed to the other processes.

Iterative methods are techniques to solve systems of equations of the form [A]{x}={b} that generate a sequence of approximations to the solution vector x. In each iteration, the coefficient matrix A is used to perform a matrix-vector multiplication. The number of iterations required to solve a system of equations with a desired precision is usually data dependent; hence, the number of iterations is not known prior to executing the algorithm. Therefore, we can analyze the performance and scalability of a single iteration of an iterative method. Iterative methods do not guarantee a solution for all systems of equations. However, when they do yield a solution, they are usually less expensive than direct methods for matrix factorization for large size of matrices.

SPARSE MATRIX COMPUTATION

BACKGROUND

We can classify matrices into two broad categories according to the kind of algorithms that are appropriate for them. The first category is *dense* or *full matrices* with few or no zero entries. The second category is *sparse matrices i*n which a majority of the entries are zero. In order to process a matrix input in parallel, we must partition it so that the partitions can be assigned to different processes. The computation and communication performed by all the message-passing programs presented so far (matrix vector and matrix-matrix algorithms for dense matrices) are quite structured. In particular, every process knows with which process it needs to communicate and what data it needs to *send* and *receive* . This is evident from matrix-matrix multiplication and matrix-vector multiplication algorithms. This information is used to map the computation onto the parallel computer and to program the required data transfers. However, there are problems in which we cannot determine a priori the processes that needs to communicate and what data they need to transfer. These problems often involve operations on irregular grid or unstructured data. Also, the exact communication patterns are specific to each particular problem and it may vary from problem to problem. Sparse matrix into vector requires unstructured communication.

Message-passing programs in order to solve these problems efficiently often need to dynamically determine the communication patterns of the algorithm. That is, the parallel program consists of two conceptual steps. The *first* step is responsible for determining which processes need to communicate with each other and what data they need to send, and the *second* step is process that performs the actual computation. The following example discuss efficient way to do sparse matrix computation using *p* processes.

SAMPLE SORT

BACKGROUND

Sorting is defined as the task of arranging an unordered collection of elements into monotonically increasing (or decreasing) order. Specifically, let $\mathbf{S} =(a_1, a_2,....., a_n)$ be a sequence of *n* elements in an arbitrary order; sorting transforms **S** into a monotonically increasing sequence $\mathbf{S}^* = (a_1^*, a_2^* ,....., a_n^*)$, such that $a_i^* <= a_j^*$ *for* $1<=i<=j<=n$, and $\mathbf{S}^*$ is a permutation of **S** . Parallelizing a sequential sorting algorithm involves distributing the elements to be sorted onto the available processes. This procedure raises a number of issues such as *where the input and output sequences are stored* and *How comparisons are performed ?.* One has to address in order to make the presentation of parallel sorting algorithms effectively.

Sorting is one of the most common operations performed by a computer. Because sorted data are easier to manipulate than randomly ordered data, many algorithms require sorted data. Sorting is of additional importance to parallel computing because of its close relation to the task of routing data among processes, which is an essential part of many parallel algorithms. Various sorting schemes have to choose between poor load balancing and irregular communication or multiple rounds of *all-to-all personalized* communication. Its significant requirements for interprocessor communication bandwidth and the irregular communication patterns that are typically generated have earned its inclusion in several parallel benchmarks. Parallel sorting strategies have still generally fallen into one of the *two* groups, each with its respective disadvantages. The *first* group, uses single step algorithms, so named because data is moved exactly once between processes. Examples of this include sample sort, parallel sorting by irregular sampling, and parallel sorting by partitioning. The price paid by these single step algorithms is an irregular communication scheme and difficulty with load balancing. The *other* group of sorting algorithms is the multi-step algorithms such as bitonic sort which accepts multiple rounds of communication in return for better load balancing and, in some cases, regular communication.

SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS BY FINITE DIFFERENCE METHOD

BACKGROUND

A significant number of the application problems in the various scientific fields involve the solution of coupled partial differential equation (*PDE*) systems in complex geometries. Finite Difference (*FD*) methods have been used to solve the coupled *PDE* systems. The first step in the solution process is to generate structured or unstructured grid in the given domain. The second step is to obtain the numerical approximations to the derivatives and unknown functions in *PDE* at each grid point and formulate the matrix system of linear equations. The last step is to solve the matrix system of equations after imposing boundary conditions. In the *FD* method, the entities within the grid elements are simulated with respect to the influence of the other entities and their surroundings. Since many such systems are evolving with time, time forms an additional dimension for these computations. Even for a small number of grid points, a three dimensional coordinate system, and a reasonable discritized time step, most of the models involve trillions of operations.

If the two dimensional rectangular region is subdivided into *n* subdomains, each group of cells within a subdomain can be processed in parallel. A special care has to be taken to handle interface grid points on each subdomain. Message passing occurs only when approximation between subdomain interconnected boundaries is performed. The approximation for derivative expressions require interprocessor communication.

Also, suppose that a parallel explicit algorithm is to be implemented on a parallel computer then these computations involve matrix-vector products and inner products which can evaluated in parallel. This can give good performance and minimize storage requirements to solve large size of partial differential equations.

On other hand, if implicit-like algorithm is used to solve the *PDE*, then one has to solve the matrix solution of system of equations $\mathbf{A}\,x = b$ in parallel. In this case, a higher level of parallelism is obtained by treating the interface nodes as a separate entity and numbering the unknowns so that all subdomains can be processed in parallel after the interface problem. The size of the interface problem determines the efficiency of this parallel strategy.

Example 24 : Description for implementation of MPI Parallel algorithm for solution of matrix system of linear equations by *Jacobi method*

- **Objective**

Write a parallel MPI program, for solving system of linear equations [A]{x} = {b} on a p processor PARAM 10000 using *Jacobi method*.

- **Description**

The Jacobi iterative method is one of the simplest iterative techniques. The i^{th} equation of a system of linear equations [A]{x}= {b} is

$$\sum_{j=0}^{n-1} A[i,j]\,x\,[j] = b\,[i] \qquad (1)$$

If all the diagonal elements of **A** are nonzero (or are made nonzero by permuting the rows and columns of **A**), we can rewrite equation (1)

$$x\,[i] = \frac{1}{A\,[i,i]}\left(b\,[i] - \sum_{j \neq i} A\,[i,j]\,x\,[j]\right) \qquad (2)$$

The Jacobi method starts with an initial guess x_0 for the solution vector x. This initial vector x_0 is used in the right-hand side of equation (2) to arrive at the next approximation x_1 to the solution vector. The vector x_1 is then used in the right hand side of equation (2), and the process continues until a close enough approximation to the actual solution is found. A typical iteration step in the Jacobi method is

$$x_k[i] = \frac{1}{A\,[i,i]}\left(b\,[i] - \sum_{j \neq i} A\,[i,j]\,x_{k-1}[j]\right) \qquad (3)$$

We now express the iteration step of equation 3 in terms of residual r_k. Equation 3 can be rewritten as

$$x_k[i] = \frac{1}{A\,[i,i]}\left(b\,[i] - \sum_{j \neq i}^{n-1} A\,[i,j]\,x_{k-1}[j]\right) + x_{k-1}[i] \qquad (4)$$

Each process computes *n/p* values of the vector x in each iteration. These values are gathered by all the processes and each process tests for convergence. If the values have been computed upto a certain accuracy the iterations are stopped otherwise the processes use these values in the next iterations to compute a new set of values.

- **Input**

The input should be in following format.

Assume that the real matrix is of size *m* x *n* and the real vector is of size *n* . Also the number of *rows m* should be greater than or equal to number of processes *p.* Process with rank *0* reads the input matrix **A** and the vector **x** . Format for the input files are given below.

Input file 1

The input file for the matrix should strictly adhere to the following format.

#Line 1 : Number of Rows (*m*), Number of columns(*n*).
#Line 2 : (*data*) (in *row-major* order. This means that the data of second row follows that of the first and so on.)

A sample input file for the matrix (8 x 8) is given below

```
8   8
61.0   2.0   3.0   4.0   6.0   8.0   9.0   2.0
 2.0  84.0   6.0   4.0   3.0   2.0   8.0   7.0
 3.0   6.0  68.0   2.0   4.0   3.0   9.0   1.0
 4.0   4.0   2.0  59.0   6.0   4.0   3.0   8.0
 6.0   3.0   4.0   6.0  93.0   8.0   3.0   1.0
 8.0   2.0   3.0   4.0   8.0  98.0   3.0   1.0
 9.0   8.0   9.0   3.0   3.0   3.0  85.0   7.0
 2.0   7.0   1.0   8.0   1.0   1.0   7.0  98.0
```

Input file 2

The input file for the vector should strictly adhere to the following format.

#Line 1 : Size of the vector (*n*)
#Line 2 : *(data)*

A sample input file for the vector (8 x 1) is given below

```
8
95.0  116.0  96.0  90.0  124.0  127.0  127.0  125.0
```

- **Output**

Process 0 will print the results of the solution *x* of matrix/system **A*x*** = **b**.

```
1.0  1.0   1.0   1.0   1.0   1.0   1.0   1.0
```

Example 25 : Description for implementation of MPI program for solution of matrix system of linear equations by *Conjugate Gradient method*

- **Objective**

You will implement a simple parallel Conjugate Gradient Method with MPI library to solve the system of linear equations [**A**] {*x*}={*b*}. Assume that **A** is symmetric positive definite matrix.

- **Description**

The conjugate gradient method

The conjugate gradient (*CG*) method is an example of minimizing method. A real n x n matrix A is positive definite if x^T **A** *x* > {0} for any *n* x 1 real, nonzero vector *x.* For a symmetric positive definite matrix **A**, the unique vector *x* that minimizes the quadratic functional f (*x*) = (1/2)x^T**A***x*-x^T*b* is the solution to the system **A*x*** = **b**, here *x* and b are *n* x 1 vectors. It is not particularly relevant when n is very large, since the conjugating time for that number of iterations is usually prohibitive and the property does not hold in presence of rounding errors. The reason is that the gradient of functional f (*x*) is **A*x*** - ***b***, which is zero when f (*x*) is minimum. The gradient of a function is a *n* x 1 vector. We explain some important steps in the algorithm. An iteration of a minimization method is of the form

$$x_{k+1} = x_k + \text{tau}_k d_k \qquad (1)$$

where tau_k is a scalar step size and $d_k$is the direction vector, d_k is a descent direction for f at x. We now consider the problem of determining tau_k, given x_k and d_k, so that f(x) is minimized on the line $x = x_k + \text{tau}_k d_k$, for tau_k. The function f(x_k+ tau d_k) is quadratic in tau, and its minimization leads to the condition

$$\text{tau}_k = g_k^T g_k / d_k^T \mathbf{A} d_k, \qquad (2)$$

where g_k=$\mathbf{A}x_k$ - **b** is the gradient (residue) vector after k iterations. The residual need not be computed explicitly in each iteration because it can be computed incrementally by using its value from the previous iteration. In the $(k+1)^{th}$ iteration, the residual g_{k+1} can be expressed as follows:

$$\begin{aligned} g_{k+1} &= \mathbf{A}x_{k+1} - b \\ &= \mathbf{A}(x_k + \text{tau}_k d_k) - b \\ &= \mathbf{A}x_k - b + \text{tau}_k \mathbf{A} d_k \\ &= g_k + \text{tau}_k \mathbf{A} d_k \end{aligned} \qquad (3)$$

Thus, the only matrix-vector product computed in each iteration is $\mathbf{A}d_k$, which is already required to compute tau_k in the equation (2). If **A** is a symmetric positive definite matrix and d_1, $d_{2,...}$, d_n are direction vectors that are conjugate with respect to **A** (that is, $d_i^T \mathbf{A} d_j$=0 for all 0<n, j<=n, i!=j), then x_{k+1} in the Equation (1) converges to the solution of **A**x = bin at most n iterations, assuming no rounding errors. In practice, however, the number of iterations that yields an acceptable approximation to the solution is much smaller than n. It also makes the gradient at x_{k+1} orthogonal to search direction, i.e $d_k^T g_{k+1} = 0$. Now we suppose that the search directions are determined by an iteration of the form

$$d_{k+1} = -g_{k+1} + \text{beta}_k d_k \qquad (4)$$

where $d_0 = -g_0$ and beta_0, beta_1 , ... remain to be determined. We find the new search direction in the plane spanned by the gradient at the most recent point and previous search direction. The parameter $\text{beta}_{k+1}$is determined by following equation

$$\text{Beta}_{k+1} = g^T_{k+1} A d_k / d^T_k A d_k \qquad (5)$$

And, one can derive orthogonality relations

$$g^T_k g_l = 0 \ (l \ != k) \qquad d^T_k A d_l = 0 \ (l \ != k)$$

The derivation of the above equation (5) and orthogonality relations is beyond the scope of this document. For details please refer [ ]. Using equation (3) and orthogonality relations, the equation (5) can be further reduced to

$$\text{Beta}_{k+1} = g^T_{k+1} g_{k+1} / g^T_k g_k \qquad (6)$$

The above equations (1) to (6) lead to *CG* algorithm. The algorithm terminates when the square of the Euclidean vector norm of gradient (residual) falls below a predetermined tolerance value. Although all of the versions of the conjugate gradient method obtained by combining the formulas for g_k, Beta_k, and tau_k in various ways are mathematically equivalent, their computer implementation is not. The following version is compared with respect to computational labor, storage requirements, and accuracy. The following sequence of steps are widely accepted

1. $\text{tau}_k = g_k^T g_k / d_k^T \mathbf{A} d_k$
2. $x_{k+1} = x_k + \text{tau}_k d_k$
3. $g_{k+1} = g_k + \text{tau}_k \mathbf{A} d_k$

4. $\text{Beta}_{k+1} = g^T_{k+1} g_{k+1} / g^T_k g_k$
5. $d_{k+1} = -g_{k+1} + \text{Beta}_k d_k$

where $k = 0, 1, 2, \ldots\ldots$ Initially we choose x_0 , calculate $g_0 = Ax_0 - b$, and put $d_0 = -g_0$

The computer implementation of this algorithm is explained as follows

```
void CongugateGradient(float  x0[ ], float b [ ], float d)
{
     float   g, Delta0, Delta1, beta;
     float   temp, tau;
     int      iteration;

     iteration = 0;
     x  = x0;
     g = b;
     g = A x - g;
     Delta0 = g^T * g;

     if ( Delta0 <= EPSILON)
        return;
     d = -g;

     do {

       iteration = iteration + 1;
       temp = A * d;
       tau = Delta0 / d^T  * temp;
       x = x + tau * d;
       g = g + tau * temp;
       Delta1 = g^T  * g;

       if ( Delta1 <= EPSILON )
            break;

       beta = Delta1 / Delta0;
       Delta0 = Delta1;
       d = -g + beta * d;

     } while(Delta0 > EPSILON && Iteration < MAX_ITERATIONS);

     return;
}
```

Regarding one-dimensional arrays of size n x 1 are required for temp, g, x, d. The storage requirement for matrix A is depends upon the structure (dense, band, sparse) of the matrix. A two dimensional n x n array is the simplest structure to store matrix A. For large sparse matrix A this structure wastes a large amount of storage space, for such matrix A suitable storage scheme should be used.

- **The preconditioned conjugate gradient algorithm**

  Let C be a positive definite matrix factored in the form $C = E E^T$, and let the quadratic functional

  $$f(x) = (1/2)x^T\mathbf{A}x - x^Tb + C ,$$

  We define second quadratic functional g(y) by the transformation $y = E^T x$, i.e.,

  $$g(x) = g(E^{-T}y) = (1/2)y^T\mathbf{A}^*y - y^Tb^* + C^* \qquad \text{where } A^* = E^{-1}AE^{-T},\ b^* = E^{-1}b,\ C^* = C.$$

  Here A^* is symmetric and positive definite. The similarity transformation

  $$E^{-T}A^*E^T = E^{-T}E^{-1}A = C^{-1}A$$

reveals that A^* and A have same eigen values. If C can be found such that the condition number of the matrix A^* is less than the condition number of the matrix A, then the rate of convergence of the preconditioned method is better than that of conjugate gradient method. We call C the *preconditioning* matrix, A^* the *preconditioned* matrix, We assume that the matrix $C = EE^T$ is positive definite, since E is nonsingular by assumption. If the coefficient matrix A has *l* distinct eigen values, the CG algorithm given converges to the solution of the system $\mathbf{A}\boldsymbol{x} = \mathbf{b}$ in at most *l* iterations (assuming no rounding errors). Therefore, if A has many distinct eigen values that vary widely in magnitude, the CG algorithm may require a large number of iterations to converge to an acceptable approximation to the solution.
The speed of convergence of the *CG* algorithm can be increased by preconditioning **A** with the congruence transformation $\mathbf{A^*} = E^{-1}AE^{-T}$ where E is a nonsingular matrix. E is chosen such that **A*** has fewer distinct eigen values than **A**. The *CG* algorithm is then used to solve $\mathbf{A^* y = b^*}$, where $x = (E^T)^{-1}y$. The resulting algorithm is called the *preconditioned conjugate gradient* (*PCG*) algorithm. The step performed in each iteration of the preconditioned conjugate gradient algorithm are as follows

1. $\text{tau}_k = g_k^T h_k / d_k^T \mathbf{A} d_k$
2. $x_{k+1} = x_k + \text{tau}_k\, d_k$
3. $g_{k+1} = g_k + \text{tau}_k\, \mathbf{A} d_k$
4. $h_{k+1} = C^{-1} g_{k+1}$
5. $\text{beta}_{k+1} = g^T_{k+1} h_{k+1} / g^T_k h_k$
6. $d_{k+1} = -h_{k+1} + \text{beta}_{k+1} d_k$

where $k = 0, 1, 2, ...$ Initially we choose x_0 , calculate $g_0 = Ax_0 - b$, $h_0 = C^{-1}g_0$ and $d_0 = -h_0$. The multiplication by C^{-1} in step (4) is to be interpreted as solving a system of equations with coefficient matrix C. A source of preconditioning matrices is the class of stationary iterative methods for solving the system $\mathbf{A}\boldsymbol{x}^* = \mathbf{b}$.

- **Parallel implementations of the PCG algorithm**

The parallel conjugate gradient algorithm involves the following type of computations and communications

Partitioning of a matrix : The matrix A is obtained by discretization of partial differential equations by finite element, or finite difference method. In such cases, the matrix is either sparse or banded. Consequently, the partition of the matrix onto *p* processes play a vital role for performance. For, simplicity , we assume that **A** is symmetric positive definite and is *rowwise* block-striped partitioned.

Scalar Multiplication of a vector and addition of vectors :
Each of these computations can be performed sequentially regardless of the preconditioner and the type of coefficient matrix. If all vectors are distributed identically among the processes, these steps require no communication in a parallel implementation.

Vector inner products :
In some situations, partial vectors are available on each processes. MPI Collective library calls are necessary to perform vector inner products If the parallel computer supports fast reduction operations, such as optimized MPI, then the communication time for the inner-product calculations can be made minimum.

Matrix-vector multiplication :
The computation and the communication cost of the matrix-vector multiplication depends on the structure of the matrix A. The parallel implementation of the PCG algorithm for three cases one in which A is a block-tridiagonal matrix of the type, two in which it is banded unstructured sparse matrix, and three in which the matrix is sparse give different performance on parallel computers. Various parts of the algorithm in each of the three cases dominate in terms of communication overheads.

Solving the preconditioned system :
The PCG algorithm solves a system of linear equations in each iteration The preconditioner C is chosen so that solving the system modified system is in expensive compared to solving the original system of equations $\mathbf{A}\boldsymbol{x} = \mathbf{b}$. Nevertheless, preconditioning increases the amount of computation in each iteration. For good preconditioners, however, the increase is compensated by a reduction in the number of iterations required to achieve acceptable convergence. The computation and the communication requirements of this step depends on the type of preconditioner used. preconditioning method such as diagonal preconditioning, in which the preconditioning matrix C has nonzero elements only along the principle diagonal does not involve any communication Also, Incomplete Cholesky (IC) preconditioning, in which C is based on incomplete Cholesky factorization of A and it may involve different computations and communications in parallel implementation.

The convergence of CG method iterations performed by checking the error criteria i.e. eulicidean norm of the residual vector should be less than prescribed tolerance. This convergence check involves gathering of real value from all processes, which may be very costly operation.

We consider parallel implementations of the PCG algorithm using diagonal preconditioner for dense coefficient matrix type. As we will see, if C is a diagonal preconditioner, then solving the modified system does not require any interprocessor

communication. Hence, the communication time in a CG iteration with diagonal preconditioning is the same as that in an iteration of the unpreconditioned algorithm.

Thus the operations that involve any communication overheads are computation of inner products, matrix-vector multiplication and, in case of IC preconditioner solving the system.

- **Input**

The input should be in following format.

Assume that the real matrix is of size *m* x *n* and the real vector is of size *n* . Also the number of *rows m* should be greater than or equal to number of processes *p.* Process with rank *0* reads the input matrix **A** and the vector **x** . Format for the input files are given below.

Input file 1

The input file for the matrix should strictly adhere to the following format.

#Line 1 : Number of Rows (*m*), Number of columns(*n*).
#Line 2 : (*data*) (in *row-major* order. This means that the data of second row follows that of the first and so on.)

A sample input file for the matrix (8 x 8) is given below

```
8   8
61.0   2.0   3.0   4.0   6.0   8.0   9.0   2.0
 2.0  84.0   6.0   4.0   3.0   2.0   8.0   7.0
 3.0   6.0  68.0   2.0   4.0   3.0   9.0   1.0
 4.0   4.0   2.0  59.0   6.0   4.0   3.0   8.0
 6.0   3.0   4.0   6.0  93.0   8.0   3.0   1.0
 8.0   2.0   3.0   4.0   8.0  98.0   3.0   1.0
 9.0   8.0   9.0   3.0   3.0   3.0  85.0   7.0
 2.0   7.0   1.0   8.0   1.0   1.0   7.0  98.0
```

Input file 2

The input file for the vector should strictly adhere to the following format.

#Line 1 : Size of the vector (*n*)
#Line 2 : *(data)*

A sample input file for the vector (8 x 1) is given below

```
8
95.0  116.0  96.0  90.0  124.0  127.0  127.0  125.0
```

- **Output**

Process 0 will print the results of the solution *x* of matrix/system **A*x*** = **b**.

1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

Example 26 : Description for implementation of MPI program for solution of matrix system of linear equations by *Gaussian Elimination method*

- **Objective**

Write a MPI program to solve the system of linear equations [**A**]{*x*} = {*b*} using Gaussian elimination without pivoting and a back-substitution. Assume that **A** is symmetric positive definite dense matrix of size *n*. You may assume that *n* is evenly divisible by *p*.

- **Description**

The system of linear equations is given by $a_{o,o}x_o + a_{o,1}x_1 + ... + a_{o,n-1}x_{n-1} = b_o$, $a_{1,o}x_o + a_{1,1}x_1 + ... + a_{1,n-1}x_{n-1} = b_1$,
....
..... $a_{n-1,o}x_0 + a_{n-1,1}x_1 + ... + a_{n-1,n-1}x_{n-1} = b_{n-1}$.

In matrix notation, this system of linear equation is written as [**A**]{*x*}={*b*} where **A**[*i, j*] = $a_{i,j}$, *b* is an *n* x 1 vector $[b_o,b_1,...,b_{n-1}]^T$, and is the desired solution vector $[x_o,x_1,...,x_{n-1}]^T$. We will make all subsequent references to $a_{i,j}$ by **A**[*i, j*] and x_i by *x*[*i*].

The matrix A is partitioned using *block row-wise* striping. Each process has *n/p* rows of the matrix A and the vector *b*. Since only process with rank *0* performs input and output, the matrix and vector must be distributed to the other processes.

The serial Gaussian elimination algorithm has three nested loops. Several variations of the algorithm exist, depending on the order in which the loops are arranged.

```
 void Gaussian_Elimination (float A[ ][ ], float b[ ], float y[ ], int n)
{
     int     i, j, k;

    for (k = 0; k < n; k++)
    {
        for (j = k+1; j < n; j++)
          A[k][j] = A[k][j] / A[k][k]   /* assume A[k][k] not equal to 0 */
        y[k]= b[k] / A[k][k];
        A[k][k] = 1;
        for (i= k+1; i < n; i++)
            {
            for (j = k+1; j < n; j++)
               A[i][j] = A[i][j] - A[i][k] * A[k][j];
            b[i] = b[i] - A[i][k] * y[k];
            A[i][k] = 0;

        }
    }
}
```

Program 1 : A serial gaussian elimination algorithm that converts the system of linear equations Ax = b to a unit upper-triangular system Ux = y. The matrix U occupies the upper-triangular locations of A.

Program shows one variation of Gaussian elimination, which we will adopt for parallel implementation in the remainder of this section. This program converts a system of linear equations $[\mathbf{A}]\{x\}=\{b\}$ to a unit upper-triangular system $[\mathbf{U}]\{x\}=\{y\}$. We assume that the matrix **U** shares storage with **A** and overwrites the upper-triangular portion of **A**. The element **A**[k, j] computed on line 6 of above program is actually **U**[k, j]. Similarly, the element **A**[k, k] equated to **1** on line 8 is **U**[k, k]. The above program assumes that **A**[k, k] not equal to 0 when it is used as a divisor on lines 6 and 7. A typical computation of Gauss elimination procedure in the k^{th} iteration of the outer loop is shown in the Figure 17

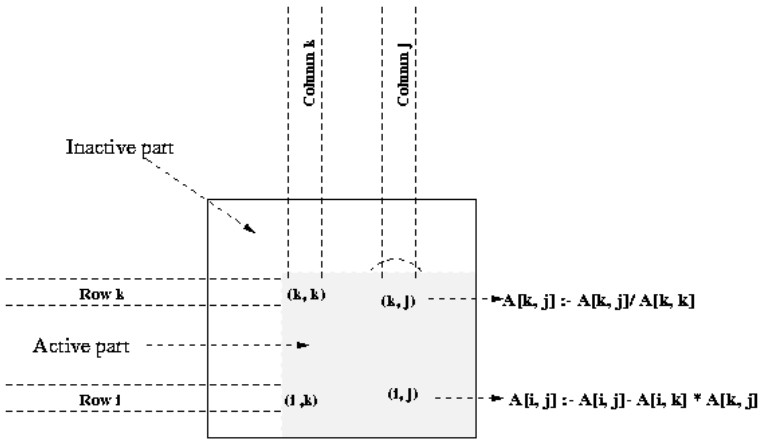


Figure 17 A typical computation in Gaussian elimination in the kth iteration

Gaussian elimination involves approximately $n^2/2$ divisions (line 6) and approximately $n^3/3$ - $n^2/2$ subtractions and multiplications (line 12). We assume that scalar arithmetic operations takes unit time. Thus, the total sequential run time of the procedure is approximately $2n^3/3$ for large n .

- **Parallel implementation with striped partitioning**

We discuss parallel formulations of the classical *Gaussian elimination* method for upper-triangularization. We describe a straight-forward *Gaussian elimination algorithm* assuming that the coefficient matrix is non singular and symmetric positive definite.

We consider a parallel implementation of program 1 in which the coefficient matrix is rowwise strip-partitioned among the processes. A parallel implementation of this algorithm with column-wise striping is very similar, and its details can be worked out based on the implementation using *rowwise striping*.

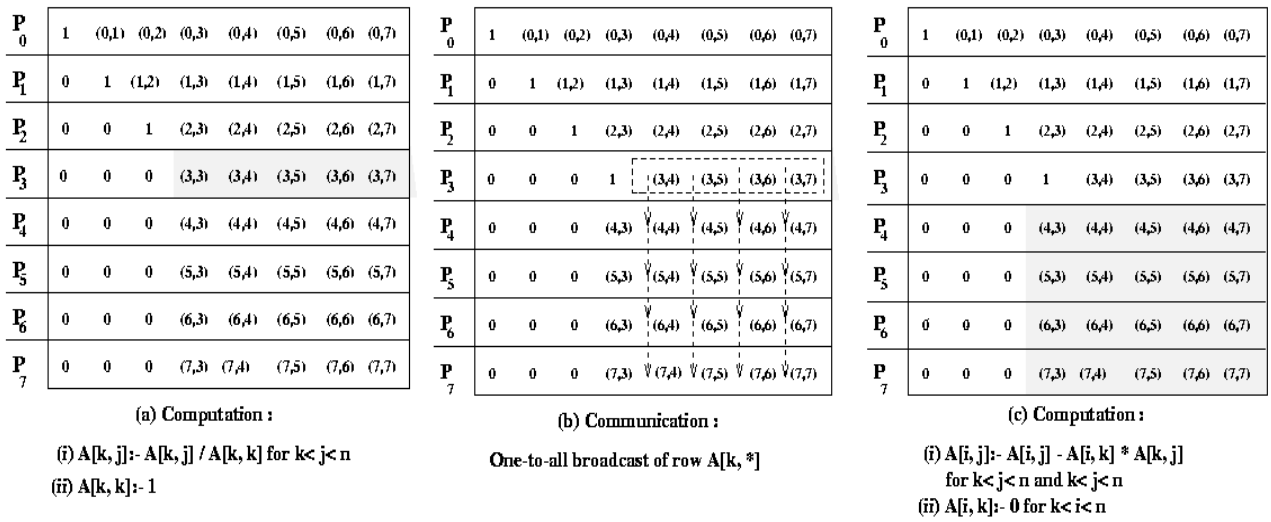


Figure 18 Gaussian elimination steps during the iteration corresponding to $k = 3$ for an 8 X 8 matrix and striped *rowwise* on 8 processors.

Figure 18 illustrates the computation and communication that takes place in the iteration of the outer loop when $k = 3$. We first consider the case in which one row is assigned to each process, and the n x n coefficient matrix A is striped among n processes labeled from P_0 to P_{n-1}. In this mapping , process P_i initially stores elements $\mathbf{A}[i, j]$ for
0 <=j <= n Figure 18(a) illustrates this mapping of the matrix onto the processors for $n = 8$.

Program 1 and Figure 18 show that $\mathbf{A}[k, k+1]$, $\mathbf{A}[k, k+2]$,..., $\mathbf{A}[k, n-1]$ are divided by $\mathbf{A}[k, k]$ (line 6) at the beginning of the k^{th} iteration. All matrix elements participating in this operation lie on the same process (shown by the shaded portion of the matrix in Figure 18(b). So this step does not require any communication. In the second computation step of the algorithm (the elimination step of line 12), the modified (after division) elements of the k^{th} row are used by all other rows of the active part of the k^{th} row to the processes storing rows $k+1$ to $n-1$. Finally, the computation
$\mathbf{A}[i, j] = \mathbf{A}[i, j] - \mathbf{A}[i, k] - \mathbf{A}[k, j]$ take place in the remaining active portion of the matrix, which is shown shaded in Figure 18(a).

This example was chosen *not because it illustrates the best way to parallelize* this particular numerical computation (it doesn't), because it illustrates the basic **MPI** *send* and *receive* operations in the context of fundamental type of parallel algorithm, applicable in many situations.

- **Input**

The input should be in following format.

Assume that the real matrix is of size m x n and the real vector is of size n . Also the number of *rows m* should be greater than or equal to number of processes *p.* Process with rank *0* reads the input matrix **A** and the vector **x** . Format for the input files are given below.

Input file 1

The input file for the matrix should strictly adhere to the following format.

#Line 1 : Number of Rows (m), Number of columns(n).
#Line 2 : (*data*) (in *row-major* order. This means that the data of second row follows that of the first and so on.)

A sample input file for the matrix (8 x 8) is given below
8 8
61.0 2.0 3.0 4.0 6.0 8.0 9.0 2.0
2.0 84.0 6.0 4.0 3.0 2.0 8.0 7.0

```
 3.0   6.0 68.0  2.0  4.0  3.0  9.0  1.0
 4.0   4.0  2.0 59.0  6.0  4.0  3.0  8.0
 6.0   3.0  4.0  6.0 93.0  8.0  3.0  1.0
 8.0   2.0  3.0  4.0  8.0 98.0  3.0  1.0
 9.0   8.0  9.0  3.0  3.0  3.0 85.0  7.0
 2.0   7.0  1.0  8.0  1.0  1.0  7.0 98.0
```

Input file 2

The input file for the vector should strictly adhere to the following format.

#Line 1 : Size of the vector (*n*)
#Line 2 : *(data)*

A sample input file for the vector (8 x 1) is given below

```
8
95.0  116.0  96.0  90.0  124.0  127.0  127.0  125.0
```

- **Output**

Process 0 will print the results of the solution *x* of matrix/system **A*x*** = **b**.

1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

Example 27 : Description for implementation of MPI program for sparse Matrix-Vector Multiplication using *block-striped partitioning*

- **Objective**

Write a MPI program on sparse matrix multiplication of size *n* x *n* and vector of size *n* on *p* processors of PARAM 10000. Assume that *n* is evenly divisible by *p* .

- **Efficient storage format for sparse matrix**

Dense matrices are stored in the computer memory by using two-dimensional arrays. For example, a matrix with *n* rows and *m* columns, is stored using a *n x m* array of real numbers. However, using the same two-dimensional array to store sparse matrices has two very important drawbacks. First, since most of the entries in the sparse matrix are zero, this storage scheme wastes a lot of memory. Second, computations involving sparse matrices often need to operate only on the non-zero entries of the matrix. Use of dense storage format makes it harder to locate these non-zero entries. For these reasons sparse matrices are stored using different data structures.

The *Compressed Storage format* (***CSR***) is a widely used scheme for storing sparse matrices. In the ***CSR*** format, a sparse matrix **A** with *n* rows having *k* non-zero entries is stored using three arrays: two integer arrays *rowptr* and *colind*, and one array of real entries *values*. The array *rowptr* is of size *n*+1, and the other two arrays are each of size *k*. The array *colind* stores the column indices of the non-zero entries in **A**, and the array *values* stores the corresponding non-zero entries. In particular, the array *colind s*tores the column-indices of the first row followed by the column-indices of the second row followed by the column-indices of the third row, and so on. The array *rowptr* is used to determine where the storage of the different rows starts and ends in the array *colind* and *values*. In particular, the column-indices of row *i* are stored starting at *colind* [*rowptr*[*i*]] and ending at (but not including) *colind* [*rowptr*[*i*+1]]. Similarly, the values of the non-zero entries of row *i* are stored at values [*rowptr*[*i*]] and ending at (but not including) values [*rowptr*[*i*+1]]. Also note that the number of non-zero entries of row *i* s simply *rowptr*[*i*+1]-*rowptr*[*i*].

	0	1	2	3	4
0	2.0		3.5		6.7
1		8.2		9.2	
2		1.1	2.8		
3	3.0		1.5	4.5	
4		2.5		8.9	

	0	1	2	3	4	5			
rowptr	0	3	5	7	10	12			

	0	1	2	3	4	5	6	7	8	9	10	11
colind	0	2	4	1	3	1	2	0	2	3	1	3

	0	1	2	3	4	5	6	7	8	9	10	11
values	2.0	3.5	6.7	8.2	9.2	1.1	2.8	3.0	1.5	4.5	2.5	8.9

Figure 19 CSR format of a sample matrix

- **Serial sparse matrix vector multiplication**

The following function performs a sparse matrix-vector multiplication [y]={**A**} {b} where the sparse matrix **A** is of size *n x* m, the vector *b* is of size *m* and the vector *y* is of size *n*. Note that the number of columns of **A** (i.e., *m*) is not explicitly specified as part of the input unless it is required.

```
void SerialSparseMatVec(int n, int *rowptr, int *colind, double *values,
                        double *b, double  *y)
{

    int i, j;

    for(i=0; i<n; i++)
    {
        y[i] = 0.0;
        for  (j=rowptr[i]; j<rowptr[i+1]; j++)
          y[i] += values [colind[j]];
    }

}
```

- **Description of parallel algorithm**

Consider the problem of computing the sparse matrix-vector product [*y*] = {**A**}{*b*} where **A** is a sparse matrix of size *m* x *n* and *b* is a dense vector using *block striped partitioning*. In the *block striped partitioning* of a matrix, the matrix is divided into groups of complete rows or columns, and each process is assigned one such group.

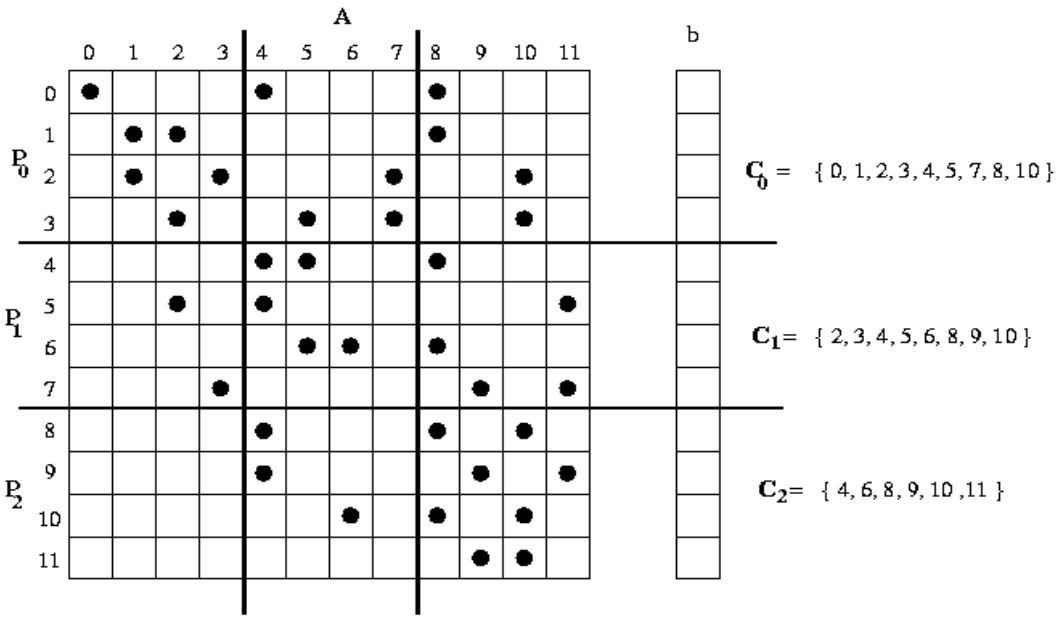


Figure 20 The data needed by each processor in order to compute the sparse matrix-vector product

In Figure 20, a sparse matrix **A** and the corresponding vector *b* are distributed among three processors. Note that processor p_0 needs to receive elements {4,5,7} from processor p_1 and elements {8,10} from processor p_2 . However, processor p_2 needs to receive only elements {4,6} from processor p_1. The process p_1 needs to receive elements {2,3} from process p_0 and elements {8,9,11} from process p_2.

Since the exact position of the non-zeros in **A** is not known a priori, and is usually different for different sparse matrices, we can not write a message-passing program that performs the required data transfers by simply hard coding. The required communication patterns may be different for different processes. That is, one process may need to receive some data from just a few processes whereas another process may need to receive data from almost all processes.

The present algorithm partitions the rows of matrix **A** using *block-striped partitioning* and the corresponding entries of vector *b* among the processes, so that each of the *p* processes gets *m/p* rows of the matrix and *n*/*p* elements of the vector. The portion of the matrix **A** obtained by *block-striped partitioning,* is assigned to each process and the non-zero entries of the sparse matrix **A** is stored using the compressed storage (*CSR*) format in the arrays *rowptr*, *colind* and *values.* To obtain the entire vector on all processes, **MPI_Allgather** collective communication is performed.

Each process now is responsible for computing the elements of the vector *y* that correspond to the rows of the matrix that it stores locally. This can be done as soon as each process receives the elements of the vector *b* that are required in order to compute these serial sparse dot-products.

This set of *b* elements depends on the position of the non-zeros in the rows of **A** assigned to each process. In particular, for each process p_i, let $\mathbf{C}_i$ be the set of column-indices *j* that contain non-zero entries overall the rows assigned to this process. Then process p_i needs to receive all the entries of the form b_j for all *j* in $\mathbf{C}_i$.

It is a simple parallel program but *inefficient* way of solving this problem because to write a message-passing program in which all the processes receive the entire *b* vector. If each row in the matrix has on the average *d* non-zero entries, then each process spends (*md/p*) time in serial algorithm. The program will achieve meaningful speedup only $d >= p$. That is, the number of non-zero entries at each row must be at least as many as the number of processes. This scheme performs well when the sparse matrices are relatively dense. However, in most interesting applications, the number of non-zero entries per row is small, often in the range of 5 to 50. In such cases, the algorithm spends more communication time relative to computation.

This example was chosen *not because it illustrates the best way to parallelize* this particular sparse numerical computations, because it illustrates the basic **MPI_Allgather** operations and ***CSR*** scheme in the context of parallel algorithm, applicable in many situations.

- **Remark**

One can design efficient algorithm by reducing communication cost by storing the necessary entries of the vector *b* . In the above algorithm, the overall communication performed by each process can be reduced if each process receives from other processes only those entries of the vector *b* are needed. In this case, we further reduce the communication cost by assigning only rows of the sparse matrix to processes such that the number of rows required but remotely stored entries of the vector *b* is minimized. This can be achieved by performing a min-cut partitioning of the graph that corresponds to the sparse matrix. We first construct graph corresponds to sparse matrix and the graph is partitioned among *p* processes. The off process communication is developed to identify the required values of the vector *b* residing on neighbouring processes.

- **Input**

The input is available in following format.

Assume that the sparse square matrix is of size *n* and is divisible by the number of processes *p* . Assume that the vector is of size *n*. For convenience, the *sparsity* is defined as maximum non-zero elements in a row, over all the rows. In the given example sparsity is 4. All the entries in the sparse matrix are floating point numbers. Process *0* reads the data. You have to adhere strictly the following format for the input files.

Input file 1

Line 1 : (*Sparsity value*)
Line 2 : (*Size of the sparse matrix*)
Line 3 : (*data* in *row-major* order) This means that the data of second row follows that of the first and so on.

A sample input file for the sparse matrix of size 16 x 16 is given below :

```
4
16
5.0  0.0  3.0  0.0  3.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  5.0  0.0
0.0  2.0  3.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  8.0  0.0  0.0  0.0  0.0
0.0  0.0  3.0  0.0  8.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  7.0  0.0  0.0
0.0  0.0  2.0  7.0  0.0  0.0  0.0  3.0  0.0  0.0  0.0  6.0  0.0  0.0  0.0  0.0
0.0  0.0  3.0  0.0  6.0  0.0  0.0  0.0  7.0  0.0  0.0  4.0  0.0  0.0  0.0  0.0
1.0  0.0  0.0  0.0  3.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  6.0  0.0
0.0  0.0  3.0  0.0  0.0  0.0  8.0  1.0  0.0  0.0  0.0  0.0  3.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  8.0  0.0  0.0  2.0  0.0  1.0  0.0  0.0  0.0  0.0  1.0  0.0
0.0  0.0  0.0  4.0  0.0  0.0  0.0  0.0  3.0  0.0  1.0  0.0  0.0  8.0  0.0  0.0
0.0  0.0  0.0  0.0  3.0  0.0  8.0  0.0  0.0  7.0  0.0  0.0  0.0  0.0  0.0  1.0
0.0  0.0  0.0  0.0  3.0  0.0  6.0  0.0  0.0  0.0  8.0  0.0  0.0  0.0  4.0  0.0
0.0  0.0  7.0  0.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  5.0  0.0  6.0  0.0  0.0
0.0  0.0  0.0  5.0  0.0  0.0  0.0  3.0  0.0  0.0  1.0  0.0  4.0  0.0  0.0  0.0
0.0  0.0  8.0  0.0  0.0  0.0  0.0  5.0  1.0  0.0  0.0  0.0  0.0  8.0  0.0  0.0
0.0  0.0  0.0  3.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  5.0  0.0  0.0  7.0  0.0
0.0  0.0  3.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  4.0  0.0  0.0  0.0  1.0
```

Input file 2

Line 1 : (*Size of the vector*)
Line 2 : (*data*)

A sample input file for the sparse vector of size 16 is given below :
16
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

- **Output**

Process with rank *0* prints the final sparse matrix-vector product.

16.0 14.0 19.0 18.0 20.0 11.0 15.0 12.0 16.0 19.0 21.0 22.0 13.0 22.0 16.0 9.0

Example 28: Description for implementation of MPI program for sorting *n* integers using *sample sort*

- **Objective**

Write a MPI program to sort *n* integers, using *sample sort* algorithm on a *p* processor of PARAM 10000. Assume *n* is multiple of *p*.

- **Description**

There are many sorting algorithms which can be parallelised on various architectures. Some sorting algorithms (bitonic sort, bubble sort, odd-even transposition algorithm, shellsort, quicksort) are based on compare-exchange operations. There are other sorting algorithms such as enumeration sort, bucket sort, sample sort that are important both practically and theoretically. First, we explain *bucket sort* algorithm and we use concepts of *bucket sort* in *sample sort* algorithm.

Serial and parallel bucket sort algorithms

Bucket sort assumes that *n* elements to be sorted are uniformly distributed over an interval (*a*, *b*). This algorithm is usually called *bucket sort* and operates as below.

The interval (*a,b*) is divided into *m* equal sized subintervals referred to as *buckets* . Each element is placed in the appropriate bucket. Since the *n* elements are uniformly distributed over the interval (*a,b)*, the number of elements in each bucket is roughly *n/m*. The algorithm then sorts the elements in each buckets, yielding a sorted sequence.

Parallelizing *bucket sort* is straightforward. Let *n* be the number of elements to be sorted and *p* be the number of processes. Initially, each process is assigned a block of *n/p* elements, and the number of *buckets* is selected to be *m=p*. The parallel formulation of *bucket sort* consists of three steps.

In the *first* step, each process partitions its block of *n/p* elements into *p* subblocks on for each of the *p* buckets. This is possible because each process knows the interval [*a,b*] and thus the interval for each bucket.

In the *second* step, each process sends subblocks to the appropriate processes. After this step, each process has only the elements belonging to the bucket assigned to it.

In the *third* step, each process sorts its bucket internally by using an optimal sequential sorting algorithm.

The performance of *bucket sort* is better than most of the algorithms and it is a good choice if input elements are uniformly distributed over a known interval.

Serial and parallel sample sort algorithms

Sample sort algorithm is an improvement over the basic *bucket sort* algorithm. The *bucket sort* algorithm presented above requires the input to be uniformly distributed over an interval [*a,b*]. However, in many cases, the input may not have such a distribution or its distribution may be unknown. Thus, using bucket sort may result in buckets that have a significantly different number of elements, thereby degrading performance. In such situations, an algorithm called *sample sort* will yield significantly better performance.

The idea behind *sample sort* is simple. A sample of size *s* is selected from the *n-element* sequence, and the range of the *buckets* is determined by sorting the sample and choosing *m*-1 elements from the results. These elements (called splitters) divide the sample into *m* equal sized buckets. After defining the buckets, the algorithm proceeds in the same way as *bucket sort* . The performance of *sample sort* depends on the sample size *s* and the way it is selected from the *n-element* sequence.

How can we parallelize the splitter selection scheme? Let *p* be the number of processes. As in *bucket sort* , set *m =p*; thus, at the end of the algorithm, each process contains only the elements belonging to a single bucket. Each process is assigned a block

of *n/p* elements, which it sorts sequentially. It then chooses *p*-1 evenly spaced elements from the sorted block. Each process sends its *p*-1 sample elements to one process say P_o. Process P_o then sequentially sorts the *p*(*p*-1) sample elements and selects the *p*-1 splitters. Finally, process P_o broadcasts the *p*-1 splitters to all the other processes. Now the algorithm proceeds in a manner identical to that of *bucket sort* . The sample sort algorithm has got three main phases. These are:

1. Partitioning of the input data and local sort :
The *first* step of sample sort is to partition the data. Initially, each one of the *p* processes stores *n/p* elements of the sequence of the elements to be sorted. Let A*i* be the sequence stored at process P_i. In the first phase each process sorts the local *n/p* elements using a serial sorting algorithm. (You can use *C* library *qsort*() for performing this local sort).

2. Choosing the Splitters :
The *second* phase of the algorithm determines the *p*-1 splitter elements *S*. This is done as follows. Each process P_i selects *p*-1 equally spaced elements from the locally sorted sequence A_i. These *p*-1 elements from these *p(p*-1*)* elements are selected to be the splitters.

3.Completing the sort :
In the *third* phase, each process P_i uses the splitters to partition the local sequence A_i into *p* subsequences $A_{i,j}$ such that for *0* $<=j <$ *p*-1 all the elements in $A_{i,j}$ are smaller than S_j , and for *j=p*-1 (i.e., the last element) $A_{i,j}$ contains the rest elements. Then each process *i* sends the sub-sequence $A_{i,j}$ to process P_j . Finally, each process sorts the received sub-sequences using merge-sort and complete the sorting algorithm.

- **Input**

  Process with rank *0* generates unsorted integers using *C* library call *rand*().

- **Output**

  Process with rank *0* stores the sorted elements in the file *sorted_data_out*.

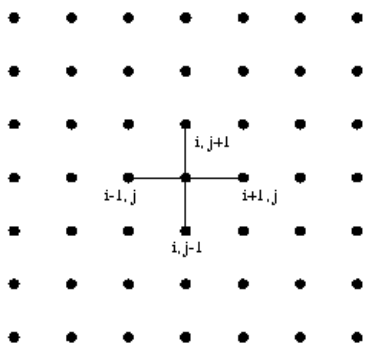
Example 29 : Description for implementation of MPI program for solution of PDE (Poisson Equation) by *finite difference method*

- **Objective**

  Write a MPI program to solve the Poisson equation with *Dirichlet* boundary conditions in two space dimensions by finite difference method on *structured rectangular type* of grid. Use *Jacobi* iteration method to solve the discretized equations.

- **Description**

  In the *Poisson problem*, the *FD* method is imposed a regular grid on the physical domain. It then approximate the derivative of unknown quantity *u* at a grid point by the ratio of the difference in *u* at two adjacent grid points to the distance between the grid point. In a simple situation, consider a square domain discretized by *n* x *n* grid points, as shown in the Figure 21(a). Assume that the grid points are numbered in a row-major order fashion from left to right and from top to bottom, as shown in the Figure 21(b). This ordering is called *natural ordering* . Given a total of n^2 points in the domain *n* x *n* grid, this numbering scheme labels the immediate neighbors of point *i* on the top, left, right, and bottom point as *i* - *n, i* -1*, i* +1 and *i*+*n* , respectively. Figure 21(b) represents partitioning of mesh using one dimensional partitioning.



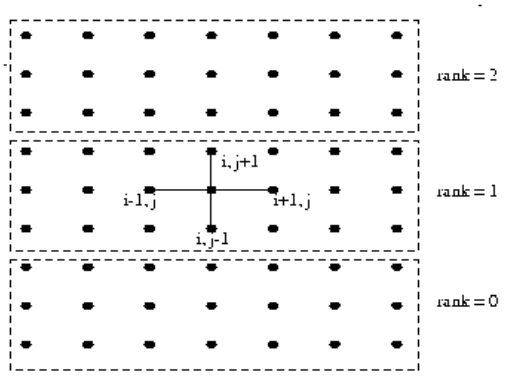


Figure 21(a) Five point stencil approximation for 2-D problem, with n=7

Figure 21(b) One dimensional decomposition of Poisson Finite Difference Grid

Formulation of Poisson Problem

The *Poisson problem* is expressed by the equations

$Lu = f(x, y)$ in the interior of domain [0,1] x [0,1]

Where L is Laplacian operator in two space dimensions.

$u(x,y) = g(x,y)$ on the boundary of the domain [0,1] x [0,1]

We define a square *mesh* (also called a *grid*) consisting of the points (x_i , y_i), given by

$x_i = i / n+1,\ i=0, ..., n+1,$

$y_j = j / n+1,\ j=0, ..., n+1,$

where there are $n+2$ points along each edge of the mesh. We will find an approximation to $u(x , y)$ only at the points (x_i, y_j). Let $u_{i, j}$ be the approximate solution to u at (x_i, y_j). and let $h = 1/(n+1)$. Now, we can approximate at each of these points with the formula $(u_{i-1, j} + u_{i, j+1} + u_{i, j-1} + u_{i+1, j} - 4u_{i, j}) / h^2 = f_{i, j}$.

Rewriting the above equation as $u_{i, j} = 1/4(u_{i-1, j} + u_{i, j+1} + u_{i, j-1} + u_{i+1, j} - h^2 f_{i, j})$,

Now jacobi iteration method is employed to obtain final solution starting with initial guess solution $u_{i,j}^{k}$ for $k=0$ for all mesh points $u_{i,j}$ and solve the following equation iteratively until the solution is converged. $u_{i, j}^{k+1} = 1/4(u_{i-1, j}^{k} + u_{i, j+1}^{k} + u_{i, j-1}^{k} + u_{i+1, j}^{k} - h^2 f_{i, j})$.

The resultant discretized banded matrix is shown in the Figure 20

- **Parallelisation strategy**

To parallelize this algorithm, the physical domain is sliced into slabs, with the computations on each slab being handled by a different process. We have used *block row-wise strip partitioning* as shown in Figure 21(a). Each process gets its own partition as shown in the Figure 21(b). To perform the computation in each partition, we need to obtain the grid point information from adjacent processs to compute the values $u_{i,j}^{k}$ for iteration k for all mesh points $u_{i,j}$ near to the partition boundary. The elements of the array that are used to hold data from other processes are called *ghost points*.

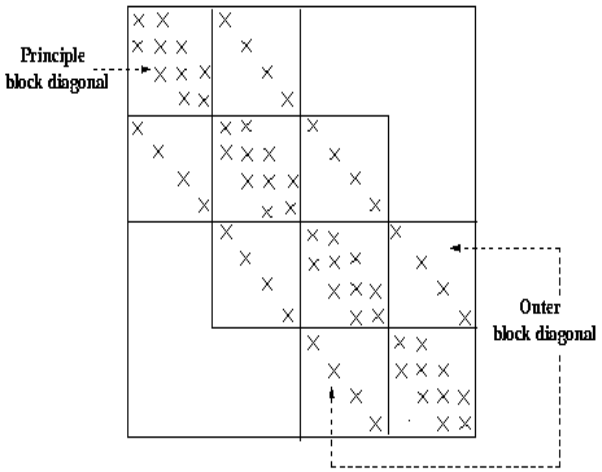


Figure 22 A 16 x 16 block-tridiagonal matrix.

In simple situation, each process sends data to the process on top and then receives data from the process below it. The order is then reversed, and data is sent to the process below and received from the process above. This strategy is simple, it is not necessarily the best way to implement the exchange of ghost points. MPI topologies can be used to create the position of processes and determine the decomposition of the arrays. MPI allows the user to define a particular application, or virtual

topology. An important virtual topology is the Cartesian topology. This simply a decomposition in the natural co-ordinate (e.g.. x,y,z) directions. In the implementation of this algorithm, row and column MPI communicators are created.

The iteration is terminated when the difference between successive computed solution at all grid points is strictly less than prescribed tolerance. The routine *MPI_Allreduce* is used to ensure that all processes compute the same value for the difference in all of the elements.

For higher order approximation, involving more than four neighbors for approximation of unknown variable or derivatives in the *PDE*, the communication at the interface of each subdomain may increase. In this case, you may require more interface information from the remote process.

More elaborate problems and algorithm often have the same communication structure that will use here to solve this problem. Thus, by studying how *MPI* can be used here, we are providing fundamentals on how communication patterns appear in more complex *PDE* problems. At the same time, we can demonstrate a wide variety of message-passing techniques and how *MPI* may be used to express them. We emphasize that while the *Poisson problem* is a useful example for describing the feature of *MPI* that can be used in solving *PDE's* and other problems that involve decomposition across many processes.

The parallel algorithm used in this model is not efficient and may give poor performance relative to more recent and sophisticated, freely available parallel solver for *PDE's* that use *MPI*. This example was *chosen* not because it illustrates the best way to parallelize this particular numerical computation (it doesn't), because it illustrates the basic *MPI send* and *receive* operations, *MPI* topologies in the context of fundamental type of parallel algorithm, applicable in many situations.

- **Input**

  User defines the total number of mesh points along each side of the square mesh (also called as grid) and number of iterations for the convergence of the method on command line. There are n+2 grid points on each side of the square mesh. Process with rank 0 performs all the input such as grid point neighbors, co-ordinate values of the nodes, and number of boundary nodes, Dirichlet boundary condition information on boundary nodes, initial guess solution on process with rank 0 is given.

  The program automatically generates all necessary input data and the user has to specify total number of grid points. To make the program as simple as possible to follow, error conditions are not checked in MPI library calls. In general one should check these, and if the calls fails, the program needs to exist smoothly.

  Various special MPI library calls can be used for the solution of poission equation by finite difference method. MPI provides to the programmer, good choice of decomposition depends on the details of the underlying hardware. MPI allows the user to define a particular application, or virtual topology. An important virtual topology is the cartesian topology which is simply a decompostion of a grid.

  Advanced point-to-point communication library calls such as *MPI_Send* and *MPI_Recv* (blocking communication calls), *MPI_Send* and *MPI_Recv* (ordered send and receive blocking communication calls), *MPI_SENDRECV* (Combined send and receive), *MPI_BSEND* (Buffer send), *MPI_ISEND*, *MPI_IRECV* (Non-blocking communication calls) and *MPI_Ssend* (synchronus sends) can be used in the implementation of parallel programe on PARAM 10000. One dimensional and two dimensional decomposition of mesh is used. Jacobi iterative method is employed for the iterative method.

- **Implementation using MPI advanced Point-to-Point Communications library calls**

  One Dimensional Decomposition of Mesh

  We consider square grid in a two dimensional region and assume that number of grid points in x-direction and y-direction is same. The actual message passing in our program is carried out by the MPI functions *MPI_Send* and *MPI_Recv*. The first command sends a message to a designated process. The second receives a message from a process. We have used simple MPI point-to-point communication calls *MPI_Send* and *MPI_Recv* in various programs. We study more detail about the advanced point-to-point communication library calls below.

  Consider an example in which process 0 wants to send a message to process 1, and there is some type of physical connection between 0 and 1. The message *envelope* contains at least the information such as the rank of the receiver, the rank of the sender, a tag and a communicator. A couple of natural questions might occur. What if a process one is not ready to receive it? What can process zero do? There are three possibilities.

- Process zero can stop and wait until process one is ready to receive the message,

- It can copy the message out of send buffer into some internal buffer. This buffer may be located on process zero, process one, or somewhere else.

- Return from the MPI_Send call or it can fail.

As long as there is space available to hold a copy of the message, the message passing system should provide this service to the programmer rather than making the process to stop dead in its tracks until the matching receive is called. Also, there is no information specifying where the receiving processes should store the message. Thus, until process 1 calls *MPI_Recv*, the system doesn't know where the data that *Process* 0 is sending should be stored. When *Process* 1 calls *MPI_Recv*, the system software that executes receive can see which (if any) buffered message has an envelope that matches the parameters specified by *MPI_Recv*. If there isn't a message, it waits until one arrives. If the MPI implementation does copy the send buffer into internal storage, we say that it buffers the data.

Different buffering strategies provide different levels of convenience and performance. For large applications that are already using a large amount of memory, even requiring the message-passing system to use all available memory may not provide enough memory to make this code work. Sometimes, the code runs (it does not ***deadlock***) but it does not execute in parallel.

All of these issues suggest that programmers should be aware of the pitfalls in assuming that the system will provide adequate buffering. If a system has no buffering, then *Process* 0 cannot send its data until it knows that *Process* 1 is ready to receive the message, and consequently memory is available for the data. Programming under the assumption that a system has buffering is very common (most systems automatically provide some buffering), although in MPI parlance, it is unsafe. This means that if the program is run on a system that does not provide buffering, the program will ***deadlock***. If the system has no buffering, Process 0's first message cannot be received until *Process* 1 has signaled that it is ready to receive the first send, and *Process* 1 will hang while it waits for Process 0 to execute the second send.

We will describe ways in which the MPI programmer can ensure that the correct parallel execution of a program does not depend on the amount of buffering, if any, provided by the message-passing system.

(1) Ordered Send and Receive

We discuss issues regarding MPI functions, which fail if there is no system buffering. We explain the issues regarding unsafe and provide alternative solutions. First, why are the functions unsafe? Consider the case that we have two processes similar to what we have given in Table 3.1. Then where will be a single exchange of data between process 0 and process 1? If we synchronize the processes before the exchanges we'll have approximately the following sequence of events:

Time	Process 0	Process 1	
1 2	MPI_Send to 1	MPI_Send to 0 MPI_Recv from 1	MPI_Recv from 0

Table 3.1 Process 0 sends a message to Process 1

As explained earlier, if there's no buffering, the *MPI_Send* will never return. So process 0 will wait in *MPI_Send* until process 1 calls *MPI_Recv*. We've encountered this situation and it is called as ***deadlock***. Two options are considered to make them safe either by re-organising the *send* and receive or let MPI make them safe through use of different MPI library calls.

In order to make them safe by reorganizing the *sends* and *receives*, we need to decide who will send first and who will receive first. One of the easiest ways to correct for a dependence on buffering is to order the sends and receives so that they are paired up. That is, the sends and receives are ordered so that if one process is sending to another, the destination will do receive that matches that send before doing a send of its own. Now, the ordered sequence of events is listed in the table 3.2

Time	Process 0	Process 1	
1 2	MPI_Send to 1	MPI_Recv from 0 MPI_Recv from 1	MPI_Send to 0

Table 3.2 Process 0 sends an message to Process 1

So there may be a delay in the completion of one communication, but the program will complete successfully.

(2) Combined Send and Receive

The approach of pairing the sends and receives is effective but can be difficult to implement when the arrangement of processes is complex (for example, with an irregular grid). An alternative is to use special MPI calls, not worrying about ***deadlock*** from a lack of buffering.

A simpler solution is to let MPI take care of the problem. The function *MPI_Sendrecv*, as its name implies, performs both send and receive, and it organizes them so that even in systems with no buffering the calling program won't deadlock, at least not in the way that the *MPI_Send* / *MPI_Recv* implementation deadlocks. The syntax of MPI_Sendrecv is

```
            int MPI_Sendrecv(void* send_buf, int send_count, MPI_Datatype send_type,
int destination, int send_tag, void* recv_buf, int recv_count, int recv_type, int
source, int recv_tag,
                             MPI_Comm comm, MPI_Status *status)
```

Notice that the parameter list is basically just a concatenation of the parameter lists for the *MPI_Send* and *MPI_Recv*. The only difference is that the communicator parameter is not repeated. The destination and the source parameters can be the same. The "send" in an *MPI_Sendrecv* can be matched by an ordinary *MPI_Recv*, and the "receive" can be matched by an ordinary

MPI_Send. The basic difference between a call to this function and *MPI_Send* followed by *MPI_Recv* (or vice versa) is that MPI can try to arrange that no deadlock occurs since it knows that the sends and receives will be paired.

Recollect that MPI doesn't allow a single variable to be passed to two distinct parameters if one of them is an output parameter. Thus, we can't call *MPI_Sendrecv* with *send buf - recv buf*. Since it is very common in practice of paired send/receives to use the same buffer, MPI provides a variant that does allow us to use a single buffer:

```
            int MPI_Sendrecv_replace(void* buffer, int count, MPI_Datatype Datatype,
int destination,
                                     int send_tag, int recv_tag, MPI_Comm comm,
MPI_Status *status)
```

Note that this implies the existence of some system buffering.

(3) Buffered Send and Receive

Instead of requiring the programmer to determine a safe ordering of the send and receive operations, MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least left in the buffer). The change in the exchange routine is simple; one just replaces the *MPI_Send* calls with *MPI_Bsend*. If the programmer allocates some memory (buffer space) for temporary storage on the sending processor, we can perform a type of non-blocking send. The semantics of buffered send mode are :

```
            int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int
dest,
                                   int tag, MP I_Comm comm, MPI_Request *request)
```

Builds a handle for a buffered send

```
              int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
                            int tag, MPI_Comm comm)
```

Basic send with user-specified buffering

In addition to the change in the exchange routine, MPI requires that the programmer provide the storage into which the message may be placed with the routine *MPI_Buffer_attach*. This buffer should be large enough to hold all of the messages.

(4) Non-blocking Send and Receive

Another approach involves using communications operations that do not block but permit the program to continue to execute instead of waiting for communications to complete. However, the ability to overlap computation with communication is just one reason to consider the non-blocking operations in MPI; the ability to prevent one communication operation from preventing others from finishing is just as important. Non-blocking communication is explicitly designed to meet these needs.

A non-blocking SEND or RECEIVE does not wait for the message transfer to actually complete, but it returns control back to the calling process immediately. The process now is free to perform any computation that does not depend on the completion of the communication operation. Later in the program, the process can check whether or not a non-blocking SEND or RECEIVE has completed and if necessary wait until it completes. The syntax of non-blocking communications has been explained in the next section. A call to a non-blocking send or receive simply starts, or posts, the communication operation. It is then up to the user program to explicitly complete the communication at some later point in the program. Thus, any non-blocking operation requires a minimum of two function calls: a call to start the operation and a call to complete the operation.

The basic functions in MPI for starting non-blocking communications are *MPI_Isend* and *MPI_Irecv*. The "I" stands for "immediate," i.e., they return (more or less) immediately. Their syntax is very similar to the syntax of *MPI_Send* and *MPI_Recv* :

```
             int MPI_Isend (void* buffer, int count, MPI_Datatype datatype int
destination, int tag,
                            MPI_Comm comm, MPI_Request* request)
```

and

```
             int MPI_Irecv (void* buffer, int count, MPI_Datatype datatype int
source, int tag,
                            MPI_Comm comm, MPI_Request* request)
```

The parameters that they have in common with *MPI_Send* and *MPI_Recv* have the same meaning. However, the semantics are different. Both calls only start the operation. For *MPI_Isend* this means this means that the system has been informed that it can start copying data out of the send buffer (either to a system buffer or to the destination). For *MPI_Irecv*, it means that the

system has been informed that it can start copying data into the buffer. Neither send nor receive buffers should be modified until the operations are explicitly completed or cancelled.

The *request* parameters purpose is to identify the operation started by the nonblocking call. So it will contain information such as the source or destination, the tag, the communicator, and the buffer. When the nonblocking operation is completed, the request initialized by the call to *MPI_Isend* or *MPI_Irecv* is used to identify the operation to be completed.

There are a variety of functions that MPI uses to complete nonblocking operations. The simplest of these is *MPI_Wait*. It can be used to complete any nonblocking operation.

```
int MPI_Wait(MPI_Request* request,MPI_Status* status)
```

The request parameter corresponds to the request parameter returned by *MPI_Isend* or *MPI_Irecv*. MPI_Wait blocks until the operation identified by request completes, if it was a *send,* either the message has been sent or buffered by the system, if it was a *receive*, the message has been copied into the receive buffer. When *MPI_Wait* returns, request is set to *MPI_REQUEST_NULL*. This means that there is no pending operation associated to *MPI_Request*. If the call to *MPI_Wait* is used to complete an operation started by *MPI_Irecv*, the information returned in the status parameter is the same as the information returned in the status by a call to *MPI_Recv*.

Finally, it should be noted that it is perfectly legal to match blocking operations with non-blocking operations. For example, a message sent with *MPI_Isend* can be received by a call to *MPI_Recv*.

(5) Communication Modes

We discussed the idea of safety in MPI. A program is safe if it will produce correct results even if the system does not provide buffering. Most programmers of message-passing systems expect the system to provide some buffering and, as a consequence, they routinely write unsafe programs. If we are writing a program that must absolutely be portable to any system, we can guarantee the safety of our program in two ways :

- We can reorganize our communications so that the program will not deadlock if sends cannot complete until a matching receive is posted.

- Other solution is to reorganize our own buffering. In either case, we are, effectively, changing the communication mode.

There are four communication modes in MPI : standard, buffered, synchronous, and ready. They correspond to four different types of send operations. There is only a standard mode for receive operations. In standard mode, it is up to the system to decide whether messages should be buffered.

Synchronous Mode

In synchronous mode a send won't complete until a matching receive has been posted and the matching receive has begun reception of the data. To ensure that a program does not depend on buffering, MPI provides the routine *MPI_Ssend*. In many special applications, it is possible to show that if the program runs successfully with no buffering, it will run with any amount of buffering. MPI provides a way to send a message in such a way that the send does not return until the destination begins to receive the message. MPI provides three synchronous modes send operations:

```
int MPI_Ssend(void* buffer, int count, MPI_Datatype datatype, int
destination,
                        int tag, MPI_Comm comm)

int MPI_Issend(void* buffer, int count, MPI_Datatype datatype, int
destination,
                         int tag, MPI_Comm comm, MPI_Request* request)

int MPI_Ssend_init(void* buffer, int count, MPI_Datatype datatype, int
destination,
                              int tag, MPI_Comm comm, MPI_Request* request)
```

Their effect is much the same as the corresponding standard mode sends. However, *MPI_Ssend* and the waits corresponding to *MPI_Issend* and *MPI_Ssend_init* will not complete until the corresponding receives have started. Thus, synchronous mode sends require no system buffering, and we can assure that our program is safe if it runs correctly using only synchronous mode sends.

Ready Mode

On some systems it's possible to improve the performance of a message transmission if the system knows, before a send has been initiated, that the corresponding receive has already been posted. For such systems, MPI provides the ready mode. The parameter lists of the ready sends are identical to the parameter lists for the corresponding standard sends:

```
int MPI_Rsend(void* buffer, int count, MPI_Datatype datatype, int
destination,
                        int tag, MPI_Comm comm)

int MPI_Irsend(void* buffer, int count, MPI_Datatype datatype, int
destination,
                         int tag, MPI_Comm comm, MPI_Request* request)

int MPI_Rsend_init(void* buffer, int count, MPI_Datatype datatype, int
destination,
                              int tag, MPI_Comm comm, MPI_Request* request)
```

The only difference between the semantics of the various ready sends and the corresponding standard sends is that the ready sends are erroneous if the matching receive hasn't been posted, and the behavior of an erroneous program is unspecified.

We have discussed in detail about the use of various MPI point-point communications library calls in order to write safe MPI programs. The different MPI library calls used in each iteration of Jacobi method may give different performance on a given parallel computer. For more details, refer the book *Pacheco*(1997) and *William*(1994).

We emphasize that while the *Poisson problem* is a useful example for describing the feature of MPI that can be used in solving *PDE*'s and other problems that involve decomposition across many processes. The parallel algorithm used in this model is not efficient and may give poor performance relative to more recent and sophisticated, freely available parallel solver for *PDE*'s that use MPI.

This example was chosen not because it illustrates the best way to parallels this particular numerical computation (it doesn't), because it illustrates the basic MPI *send* and *receive* operations, MPI topologies in the context of fundamental type of parallel algorithm, applicable in many situations.

Two Dimensional Decomposition of Mesh

We use higher dimensional decompositions to partition the domain and find the solution to the Poisson problem by using Jacobi iteration. We let MPI to compute the decomposition of the domain using *MPI_Cart_create*.

In the earlier programs, the messages have consisted of contiguous areas in memory, so the basic datatypes such as *MPI_INTEGER* and *MPI_DOUBLE_PRECISION*, accompanied by a count, have been sufficient to describe the messages. We use MPI's derived datatypes, which allow us to specify noncontiguous direction.

In the earlier iteration scheme, we have used *MPI_Send* and *MPI_Recv* to exchange data between top and below processes. This strategy is simple but it may pose problems if the parallel system doesn't have adequate buffer. Consequently, it is not necessarily the best way to implement the exchange of ghost points. The simplest solution to correct for a dependence on buffering is to order the sends and receives so that they are paired up. That is, the sends and receives are ordered so that if one process is sending to another, the destination will do receive that matches that send before doing a send of its own.

The approach of pairing the sends and receives is effective, but can be difficult to implement when the arrangement of processes is complex. An alternative is to use the *MPI_Sendrecv*. This routine allows you to send and receive data without worrying about deadlock from a lack of buffering. Each process then sends to the process below it and receives from the process above it.

- **Output**

The program prints the progress of the iterations, final iteration count, computed solution and the difference of the solution on process with rank *0*.

Contents Hands-on Session MPI Library Calls MPI On Web