HPC PROJECT

ASHMITHA BONAVENTURE.M

DATA FORMAT – SPARSE & SYMMETRIC MATRIX

- SPARSE : A **sparse matrix** is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered **dense**.

Example of sparse matrix

$$\begin{pmatrix} 11 & 22 & 0 & 0 & 0 & 0 & 0 \\ 0 & 33 & 44 & 0 & 0 & 0 & 0 \\ 0 & 0 & 55 & 66 & 77 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 88 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 99 \end{pmatrix}$$

The above sparse matrix contains only 9 nonzero elements, with 26 zero elements. Its sparsity is 74%, and its density is 26%.

- Sparse data is by nature more easily compressed and thus require significantly less storage.

- SYMMETRIC : A symmetric matrix is a square matrix that is equal to its transpose : A=Transpose(A)

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix}^{T}$$

SPARSE & SYMMETRIC MATRIX – EXAMPLE

- A symmetric sparse matrix is a matrix where a(i, j) = a(j, i) for all i and j. Because of this symmetry, only the lower triangular values need to be passed to the solver routines. The upper triangle can be determined from the lower triangle.

$$A = \begin{bmatrix} 4.0 & 1.0 & 2.0 & 0.5 & 2.0 \\ 1.0 & 0.5 & 0.0 & 0.0 & 0.0 \\ 2.0 & 0.0 & 3.0 & 0.0 & 0.0 \\ 0.5 & 0.0 & 0.0 & 0.625 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & 16.0 \end{bmatrix}$$

- CSC Format : colptr - 1, 3, 6, 7, 9 ; rowind: 1, 2, 1, 2, 4, 3, 2, 4; values: 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0

CSR – DATA FORMAT

- When multiplying a large n × n sparse matrix A having nnz non-zeros by a dense n-vector x, the memory bandwidth for reading A can limit overall performance.
- The current standard storage for- mat for sparse matrices in scientific computing, compressed sparse rows (CSR), is more efficient, because it stores only n + nnz indices or pointers.
- Parallel procedure for computing y ← Ax where nXn matrix A is stored in CSR format.

1. n←A.rows
2. for i←0 to n−1 in parallel
3. do y[i] ← 0
4. for k ← A. row_ptr[i] to A. row_ptr[i + 1] − 1
5. do y[i] ← y[i] + A. val[k] · x[A. col_ind[k]]

SAMPLE DATA

```
T:
triplet: 4-by-4, nzmax: 16 nnz: 10
    2 2 : 3
    1 0 : 1
    3 3 : 1
    0 2 : 3
    1 1 : 2
    3 0 : 3
    3 1 : 1
    1 3 : 1
    0 0 : 2
    2 1 : 1
```

```
A:
4-by-4, nzmax: 10 nnz: 10, 1-norm: 6
    col 0 : locations 0 to 2
      1 : 1
      3 : 3
      0 : 2
    col 1 : locations 3 to 5
      1 : 2
      3 : 1
      2 : 1
    col 2 : locations 6 to 7
      2 : 3
      0 : 3
    col 3 : locations 8 to 9
      3 : 1
      1 : 1
```

```
col 0 : locations 0 to 0
  0 : 2
col 1 : locations 1 to 1
  1 : 2
col 2 : locations 2 to 2
  2 : 2
col 3 : locations 3 to 3
  3 : 2
```

- The input matrix(T) is given in format (row_num, col_num : value) nzmax - max number of nonzero in matrix, nnz- number of non-zeroes fed.
- The T matrix is then transformed to A matrix according to the column and the row-wise.
- The last is the vector in the example used it is 2. The final matrix gets also stored in the same format as the input.

MPI

- MPI is a library of routines that can be used to create parallel programs in C or Fortran77.
- MPI is a library that runs with C or Fortran programs, using commonly-available operating system services to create parallel processes and exchange information among these processes.
- MPI is designed to allow users to create programs that can run efficiently on most parallel architectures.
- MPI can also support distributed program execution on heterogenous hardware. That is, we may run a program that starts processes on multiple computer systems to work on the same problem. This is useful with a workstation farm.

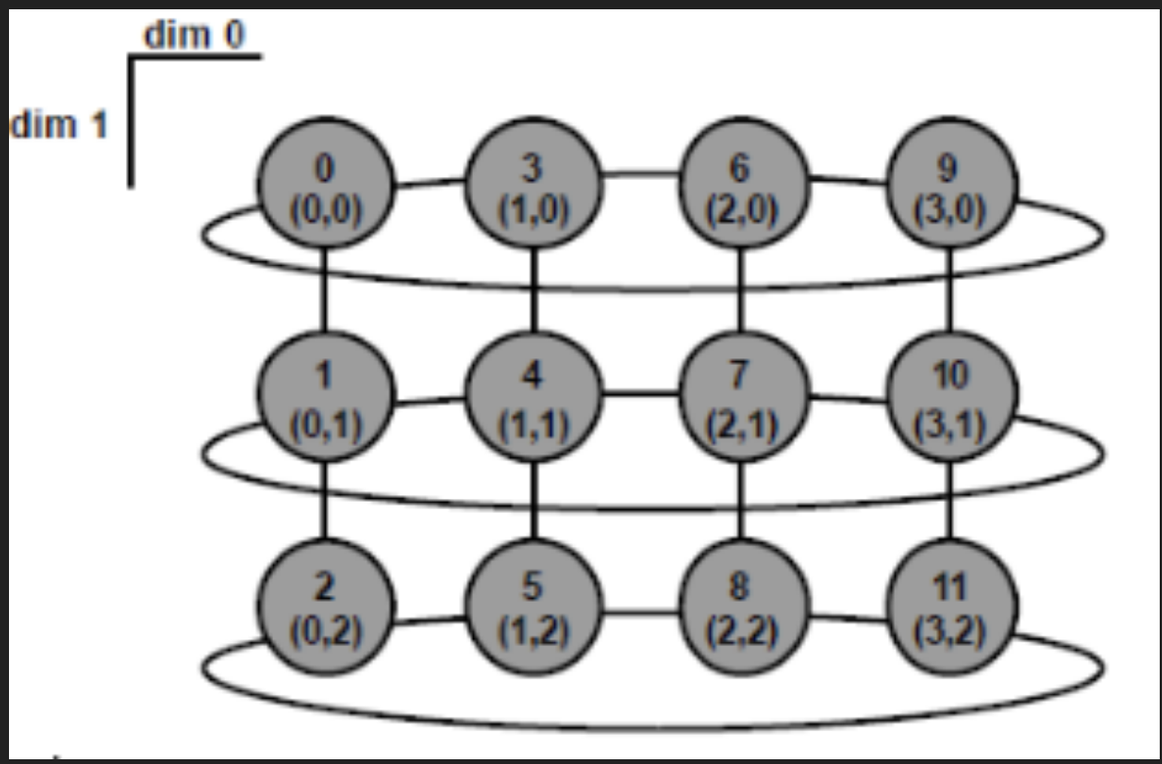
DATA DISTRIBUTION BETWEEN PROCESSORS

- When the program starts, it consists of only one process, sometimes called the "parent", "root", or "master" process.
- When the routine MPI_Init executes within the root process, it causes the creation of 3 additional processes
- Each of the processes then continues executing separate versions of the hello world program.
- MPI assigns an integer to each process beginning with 0 for the parent process and incrementing each time a new process is created.
- The process numbers are not printed in ascending order.

```
#include <stdio.h>
  #include <mpi.h>
   main(int argc, char **argv)
   {
     int ierr, num_procs, my_id;
     ierr = MPI_Init(&argc, &argv);
     .
     ierr =
MPI_Comm_rank(MPI_COMM_WORLD,
&my_id);
     ierr =
MPI_Comm_size(MPI_COMM_WORLD,
&num_procs);
     printf("Hello world! I'm process %i out
of %i processes\n",
       my_id, num_procs);
     ierr = MPI_Finalize();
   }
```

CARTESIAN TOPOLOGY

- Topology is one of the attributes for communicator. In MPI, a topology is a mechanism for associating different addressing schemes with the processes belonging to a group.
- MPI topology is a virtual topology: there is no simple relation between the process structure and actual underlying physical structure of the parallel system.
- Two main topology types: Cartesian (or grid) and graphs. Graphs are the more general case.
- EXAMPLE ->



CREATING A CARTESIAN VIRTUAL TOPOLOGY

- int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart);

comm_old: input communicator

• ndims: number of dimensions of cartesian grid

• dims: integer array of size ndims specifying the number of processes in each dimension

• periods: logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension

• reorder: ranking may be reordered (true) or not (false) (logical)

• comm_cart[out]: communicator with new cartesian topology (handle)

```
MPI_Comm grid_comm;
int dim_sizes[2];

int wrap_around[2]; int
reorder = 1;

dim_sizes[0] = 4;
dim_sizes[1] = 3;

wrap_around[0] = 1;
wrap_around[1] = 0;

MPI_Cart_create
(MPI_COMM_WORLD, 2,
dim_sizes,wrap_around,
reorder, &grid_comm);
```

- Create 2D Cartesian topology for processes

  MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder, &comm2D);

- For Finding the *NEARBY* neighbours rank &

for a process to determine its coordinates.

0,0 (0)	0, 1 (1)
1, 0 (2)	1, 1 (3)
2, 0 (4)	2, 1 (5)

MPI_Comm_rank(comm2D, &id2D);

MPI_Cart_coords(comm2D, id2D, ndim,

coords2D);

int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *source, int *dest);

SEQUENTIAL MATRIX MULTIPLICATION

- The sequential algorithm of multiplying matrix by vector may be represented in the following way.

```
for (i = 0; i < m; i++)  { c[i] = 0;

 for (j = 0; j < n; j++) {

   c[i] += A[i][j]*b[j]  } }
```

- Input data – A[m][n] – matrix of order m×n ; b[n] – vector of n elements.
- Result – c[m] – vector of m elements.
- Inner multiplication of vectors of size n, time complexity is the order O(n), to execute m operations of inner multiplication. Thus, the algorithm's time complexity is the order O(mn).

EXAMPLE

The general formula for a matrix-vector product is

$$\mathbf{Ax} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}$$

- Matrix_multiply method concept :

```
void MatrixMulVec (double **a, double *b, double *r, int M, int N) {
    for (int i = 0; i < M; i++) {
        r[i] = 0;
        for (int j = 0; j < N; j++) {
            r[i] += a[i] [j] * x[j] ;  }}}
```

EXAMPLE – CNTD.

```
A:
4-by-4, nzmax: 10 nnz: 10, 1-norm: 6
    col 0 : locations 0 to 2
      1 : 1
      3 : 3
      0 : 2
    col 1 : locations 3 to 5
      1 : 2
      3 : 1
      2 : 1
    col 2 : locations 6 to 7
      2 : 3
      0 : 3
    col 3 : locations 8 to 9
      3 : 1
      1 : 1
```

vector ->

```
col 0 : locations 0 to 0
  0 : 2
col 1 : locations 1 to 1
  1 : 2
col 2 : locations 2 to 2
  2 : 2
col 3 : locations 3 to 3
  3 : 2
```

A*AT = C

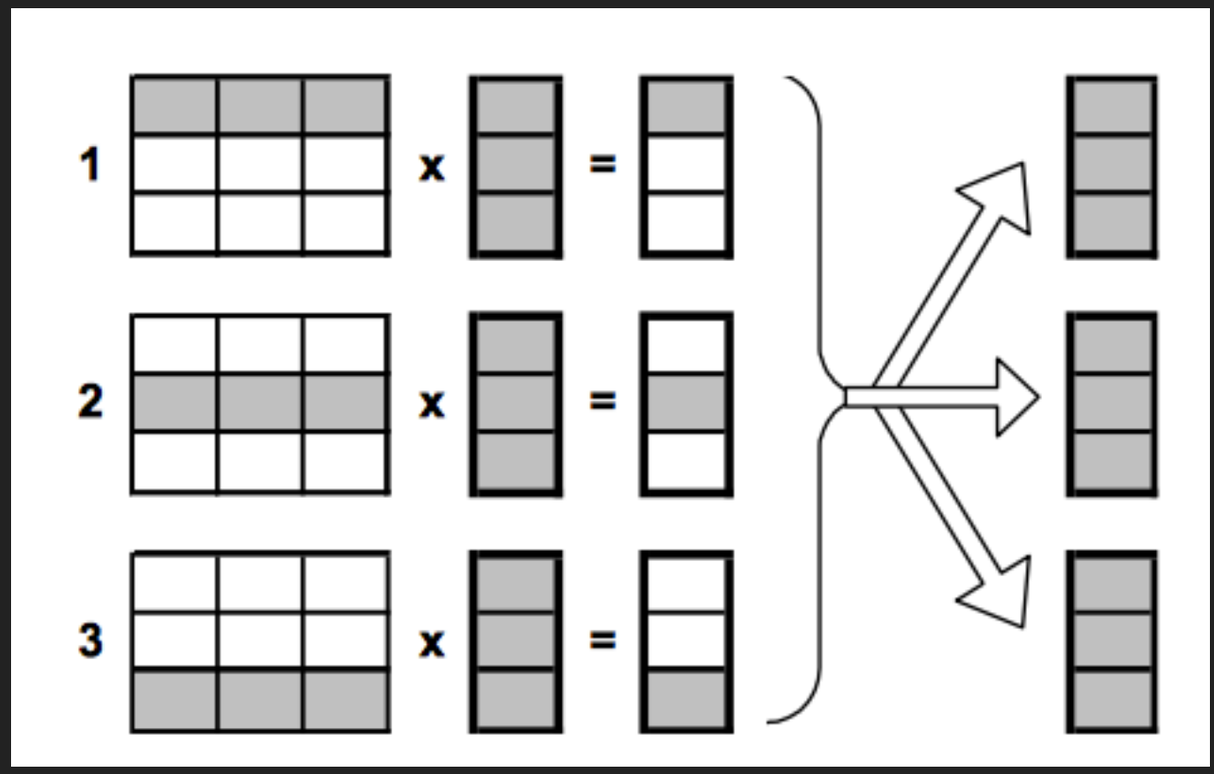
```
AT:
4-by-4, nzmax: 10 nnz: 10, 1-norm: 5
    col 0 : locations 0 to 1
      0 : 2
      2 : 3
    col 1 : locations 2 to 4
      0 : 1
      1 : 2
      3 : 1
    col 2 : locations 5 to 6
      1 : 1
      2 : 3
    col 3 : locations 7 to 9
      0 : 3
      1 : 1
      3 : 1
```

=

```
C:
4-by-4, nzmax: 16 nnz: 16, 1-norm: 30
    col 0 : locations 0 to 3
      1 : 2
      3 : 6
      0 : 13
      2 : 9
    col 1 : locations 4 to 7
      1 : 6
      3 : 6
      0 : 2
      2 : 2
    col 2 : locations 8 to 11
      1 : 2
      3 : 1
      2 : 10
      0 : 9
    col 3 : locations 12 to 15
      1 : 6
      3 : 11
      0 : 6
      2 : 1
```

PARALLEL MATRIX MULTIPLICATION

- Let us consider the algorithm of matrix-vector multiplication, which is based on row-wise block-striped matrix decomposition scheme.
- To execute the basic subtask of inner multiplication the processor must contain the corresponding row of matrix A and the copy of vector b.



- After computation completion each basic subtask determines one of the elements of the result vector c.
- To combine the computation results and to obtain the total vector c on each processor of the computer system, it is necessary to execute the all gather operation(MPI_Allgather of MPI library)

STEPS

- Get the dimensions :

  MPI_Bcast(m_p, 1, MPI_INT, 0,comm);

  MPI_Bcast(n_p, 1, MPI_INT, 0, comm);

- Read the matrix and the vector :

  MPI_Scatter(A, local_m*n, MPI_DOUBLE, local_A, local_m*n, MPI_DOUBLE, 0, comm);

  MPI_Scatter(vec, local_n, MPI_DOUBLE, local_vec, local_n, MPI_DOUBLE, 0, comm);

- Print the matrix and the vector :

  MPI_Gather(local_A, local_m*n, MPI_DOUBLE, A, local_m*n, MPI_DOUBLE, 0, comm);

  MPI_Gather(local_vec, local_n, MPI_DOUBLE, vec, local_n, MPI_DOUBLE, 0, comm);

- **Main :**

  Mat_vect_mult(local_A, local_x, local_y, x, m, local_m, n, local_n);

  MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm); sendbuf

FUNCTIONS

- **int MPI_Allgatherv (void * send_bufr, int send_count, MPI_Datatype send_type, void * recv_bufr, int * recv_count, int * recv_disp, MPI_Datatype recv_type, MPI_Comm comm);**

send_bufr : Starting address of send buffer

send_count : Number of elements in send buffer ; send_type : Data type of send buffer elements

recv_bufr : The only output parameter; Starting address of receive buffer to store the gathered elements

recv_count : Integer array containing the number of elements to be received from each process

recv_disp : Integer array to specify the displacement relative to recv_bufr at which to place the incoming data from processes

recv_type : Data type of receive buffer elements comm Communicator

- **int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) ;**

address of send buffer (choice) ; count - number of elements in send buffer (integer)

datatype - data type of elements of send buffer (handle) ; op - -reduce operation (handle)

root - rank of root process (integer) ; comm- communicator (handle)

CONJUGATE GRADIENT

- Conjugate gradient method is a method used to find the solution x of **b=Ax,** namely those whose matrix is symmetric.
- Starting with an initial guess of the solution, we calculate the gradient to find a first direction to move. During the next iterations, the directions have to be conjugate to the previous ones.
- Therefore, CG is particularly useful when *A* is a sparse matrix since these operations are usually extremely efficient.
- One matrix-vector multiplication per iteration, two vector dot products per iteration, four n-vectors of working storage

$$x_0 = 0, \quad r_0 = b, \quad d_0 = r_0$$

for $k = 1, 2, 3, \ldots$

$$\alpha_k = (r^T_{k-1} r_{k-1}) / (d^T_{k-1} A d_{k-1}) \qquad \text{step length}$$

$$x_k = x_{k-1} + \alpha_k d_{k-1} \qquad \text{approx solution}$$

$$r_k = r_{k-1} - \alpha_k A d_{k-1} \qquad \text{residual}$$

$$\beta_k = (r^T_k r_k) / (r^T_{k-1} r_{k-1}) \qquad \text{improvement}$$

$$d_k = r_k + \beta_k d_{k-1} \qquad \text{search direction}$$

SEQUENTIAL CG

- A chosen version of the Sequential Conjugate Gradient Algorithm (SCGA) is convergent only if the coefficient matrix of the system of linear equations is symmetric and positive definite.
- **STEPS**:
  - Get the initial residue, where A is the input matrix, x_R is the final vector, b-RHS vector, x-initial guess.

```
Read_Mat(inputMatrix, &MAT) :
for(int i=0; i<MAT->nz; i++)
  {  if (fscanf(f, "%d %d %lg\n", &MAT->IA[i], &MAT->JA[i], &MAT->val[i])==0)
        fprintf(stderr,"\nFailed to read input Matrix\n");
     MAT->IA[i]--;  /* adjust from 1-based to 0-based */
     MAT->JA[i]-- ;}
```

  - For the Input matrix read the format convert to CSR format and sort it (this format allows fast row access and matrix-vector multiplications)

```
convertToCSR(&MAT);
for(int j=1;j<MAT->n_row_ptr-1;j++) {
      for(int i=init;i<MAT->nz;i++) {
          if(MAT->IA[i+1] != MAT->IA[i]) {
              MAT->row_ptr[j] = i+1 ; init = i+1 ; break ;  }}}
MAT->row_ptr[MAT->n_row_ptr-1] = MAT->nz;
```

STEPS : CNTD.

- Initialise the solution vector and the initial guess is 0.0

```
for(int i=0;i<numColumns;i++) {
        x[i] = 1.0 ;
        init_guess_x0[i] = 0.0 ; }
```

- Vector dot product :

```
dot = a[0]*b[0];
for (i=1; i<N; i++)
dot += a[i]*b[i];
```

- Matrix vector product (r_k, MAT,init_guess_x0);

```
numIterations = numIterations + 1;
residue = pow(rk_dot_rk_new,0.5); // rk_dot_rk_new =
vectorDotProduct(r_k,r_k,dim);
relative_residue = (residue/residue_init);
```

OUTPUT

- On selecting the matrix the computation starts, the iterations get incremented by 1 and the residues and the relative residues are calculated.

```
//Calculating the total time consumed by the Conjugate gradient step

    start_time = omp_get_wtime();

    ConjugateGradient(&MAT, numColumns, x_k, b, init_guess_x0, tolerance, max_iterations);

    run_time = omp_get_wtime() - start_time;

    fprintf(stdout,"Time for computation = %lf s\n",run_time);
```

```
Matrix 1 selected
Iteration No. = 1
Initial Residue = 5.687279e+07   Current Residue = 1.338702e+07  Relative Residue = 2.353853e-01

Iteration No. = 2
Initial Residue = 5.687279e+07   Current Residue = 5.397195e+06  Relative Residue = 9.489941e-02
```

```
Iteration No. = 1748
Initial Residue = 5.687279e+07   Current Residue = 5.719945e-01  Relative Residue = 1.005744e-08

Iteration No. = 1749
Initial Residue = 5.687279e+07   Current Residue = 5.636989e-01  Relative Residue = 9.911574e-09

Number of Iterations = 1749

Initial Residue = 5.687279e+07   Final Residue = 5.636989e-01    Final Relative Residue = 9.911574e-09
Time for computation = 3.501057 s
```

PARALLEL CG

- After many numerical investigations, parallel version of the algorithm was based on an assumption that the most time consuming and possible to parallelize operations are:
  - matrix-matrix multiplication,
  - matrix-vector multiplication,
  - vector inner product.

- STEPS:
  - Get the initial residue, where A is the input matrix, x_R is the final vector, b-RHS vector, x-initial guess.
  - For the Input matrix read the format convert to CSR format and sort it (this format allows fast row access and matrix-vector multiplications)
  - Distribute matrix between processors, followed by CG.
- Matrix distribution is done by row-wise.

STEPS

Step 1. Read the data from the input file and divide it across processors, using MPI Bcast and MPI Scatter(v).

MPI_Bcast(local_vectorX,ROWS*COL,MPI_FLOAT,0,MPI_COMM_WORLD);

MPI_Scatter(matrixA,local_row*COLS,MPI_FLOAT,local_matrixA,local_row*COLS, MPI_FLOAT, 0,MPI_COMM_WORLD);

MPI_Scatter(vectorB,local_row,MPI_FLOAT,local_vectorB,local_row, MPI_FLOAT, 0,MPI_COMM_WORLD);

Step 2. Compute the inner product locally, and do a sum reduction (MPI Allreduce) across all processes.

MPI_Allreduce(&temp_a,&step_len,1,MPI_FLOAT,MPI_SUM,MPI_COMM_WORLD);

MPI_Allreduce(&temp_b,&imp_step_len,1,MPI_FLOAT,MPI_SUM,MPI_COMM_WORLD);

Step 3. Do the matrix-vector product in parallel using MPI Allgatherv, to gather all the local parts of the vector into a single vector, and then to do the multiplication.

MPI_Allgather(local_vectorP, local_row, MPI_FLOAT, vectorP, local_row, MPI_FLOAT, MPI_COMM_WORLD);

OUTPUT

- Sequential:
- Time for computation = 3.678953 s
- Parallel- Number of Processors -
- 2 : Time for computation = 3.511360 s
- 3: Time for computation = 3.567981 s
- 4: Time for computation = 3.773639 s

LU DECOMPOSITION

- Let A be a square matrix. An LU factorization refers to the factorization of A, into two factors a lower triangular matrix L and an upper triangular matrix U.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

- We transform the matrix A into an upper triangular matrix U by eliminating the entries below the main diagonal same for lower.
- It results in a unit lower triangular matrix and an upper triangular matrix.

CNTD.

- To factor the n × n matrix A = [aij] into the product of the lower-triangular matrix L = [lij] and the upper-triangular matrix U = [uij]; that is, A = LU, where the main diagonal of either L or U consists of all ones :
  - INPUT dimension n; the entries aij, 1 ≤ i,j ≤ n of A; the diagonal l11 =⋯=lnn =1 of L or the diagonal u 11 =⋯= unn =1of U.
- OUTPUT the entries lij,1≤j ≤i,1≤i ≤n of L and the entries, uij, i ≤ j ≤ n, 1 ≤ i ≤ n of U.

ALGORITHM

Step 1 Select l_{11} and u_{11} satisfying $l_{11}u_{11} = a_{11}$
If $l_{11}u_{11} = 0$ then OUTPUT ('Factorization impossible')
STOP

Step 2 For $j = 2, \ldots, n$ set $u_{1j} = a_{1j}/l_{11}$ (*First row of U*)
$l_{j1} = a_{j1}/u_{11}$ (*First column of L*)

Step 3 For $i = 2, \ldots, n-1$ do Steps 4 and 5:

Step 4 Select l_{ii} and u_{ii} satisfying $l_{ii}u_{ii} = a_{ii} - \sum_{k=1}^{i-1} l_{ik}u_{ki}$
If $l_{ii}u_{ii} = 0$ then OUTPUT ('Factorization impossible')
STOP

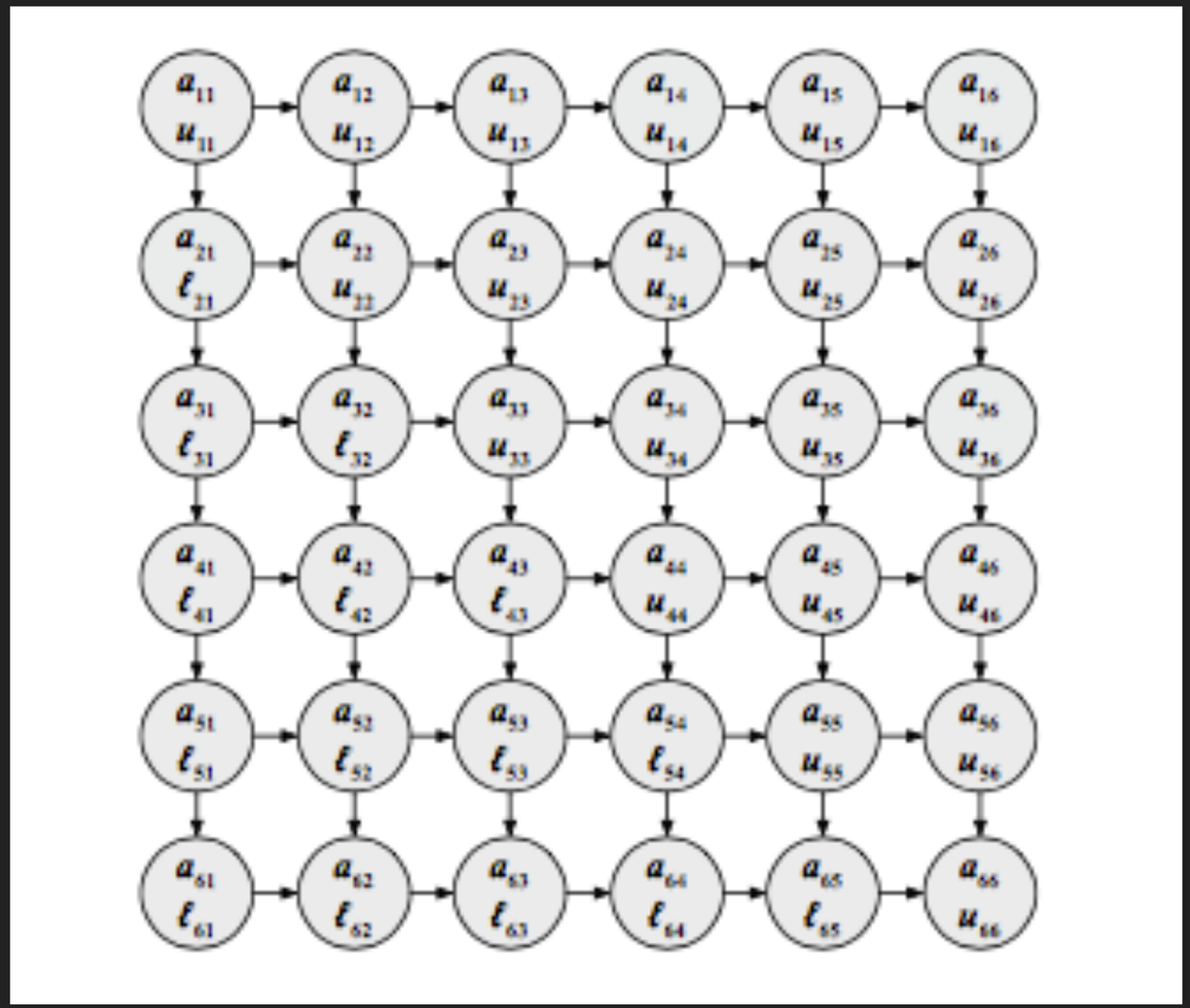
Step 5 For $j = i+1, \ldots, n$
set $u_{ij} = \frac{1}{l_{ii}}\left[a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\right]$ (*ith row of U*)
$l_{ji} = \frac{1}{u_{ii}}\left[a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki}\right]$ (*ith column of L*)

Step 6 Select l_{nn} and u_{nn} satisfying $l_{nn}u_{nn} = a_{nn} - \sum_{k=1}^{n-1} l_{nk}u_{kn}$

(*Note: If $l_{nn}u_{nn} = 0$, then $A = LU$ but A is singular*)

PARALLEL LU–FINE GRAINED TASKS & COMMUNICATION

- **Partition** :
  - For i,j = 1,...,n, fine-grain task (i,j) stores aij and computes and stores
    - uij, if i ≤ j
    - lij, if i > j



ALGORITHM

- **Communicate** :
  - Broadcast entries of A vertically to tasks below
    Broadcast entries of L horizontally to tasks to right.

for $k = 1$ **to** $\min(i, j) - 1$
  recv broadcast of a_{kj} from task (k, j) { vert bcast }
  recv broadcast of ℓ_{ik} from task (i, k) { horiz bcast }
  $a_{ij} = a_{ij} - \ell_{ik}\, a_{kj}$ { update entry }
end
if $i \leq j$ **then**
  broadcast a_{ij} to tasks $(k, j),\ k = i + 1, \ldots, n$ { vert bcast }
else
  recv broadcast of a_{jj} from task (j, j) { vert bcast }
  $\ell_{ij} = a_{ij}/a_{jj}$ { multiplier }
  broadcast ℓ_{ij} to tasks $(i, k),\ k = j + 1, \ldots, n$ { horiz bcast }
end

INCOMPLETE LU

- For a typical sparse matrix, the LU factors can be much less sparse than the original matrix. The memory requirements for using a direct solver can then become a bottleneck in solving linear systems.
- An incomplete factorization instead seeks triangular matrices L, U such that A≈≈LU rather than A=LU Solving for LU x=b can be done quickly but does not yield the exact solution to Ax=b. So, we instead use the matrix M=LU, as a pre conditioner in another iterative solution algorithm such as the conjugate gradient method.
- More accurate ILU preconditioners require more memory, to such an extent that eventually the running time of the algorithm increases even though the total number of iterations decreases.

PARALLEL INCOMPLETE LU

- The algorithm is easy to parallelize and has much more parallelism than existing approaches.
- Each nonzero of the incomplete factors L and U can be computed in parallel with an asynchronous iterative method, starting with an initial guess.
- A feature of the algorithm is that, unlike existing approaches, it does not rely on reordering the matrix in order to enhance parallelism.
- Reordering can instead be used to enhance convergence of the solver.
- When using ILU preconditioners in a parallel environment, the sparse triangular solves must also be parallelized.
- The new fine-grained parallel algorithm inter- prets an ILU factorization as, instead of a Gaussian elimination process, a problem of computing unknowns lij and uij which are the entries of the ILU factorization.

CNTD.

- Does not use level scheduling or reordering of the matrix
- Each entry of L and U updated iteratively in parallel
- Method can be used to update a factorization given a new A
- Can use asynchronous computations (helps tolerate latency and performance irregularities)

ALGORITHM–FINE-GRAINED PARALLEL INCOMPLETE FACTORIZATION

1 Set unknowns l_{ij} and u_{ij} to initial values
2 **for** *sweep* $= 1, 2, \ldots$ *until convergence* **do**
3 **parallel for** $(i, j) \in S$ **do**
4 **if** $i > j$ **then**
5 $l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}\right) / u_{jj}$
6 **else**
7 $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
8 **end**
9 **end**
10 **end**

- The algorithm is parallelized across the elements of S. Given p compute threads, the set S is partitioned into p parts, one for each thread. The threads run in parallel, updating the components of the vector of unknowns, x.Thus the latest values of x are used in the updates.

SYMMETRIC FINE-GRAINED PARALLEL INCOMPLETE FACTORIZATION.

- When the matrix A is symmetric positive definite , the algorithm need only compute one of the triangular factors. Specifically, an incomplete Cholesky (IC) factorization computes U T U ≈ A, where U is upper triangular.
- The below algorithm shows the pseudocode for this case, where we have used uij to denote the entries of U and SU to denote an upper triangular sparsity pattern.

```
1  Set unknowns u_ij to initial values
2  for sweep = 1, 2, ... until convergence do
3      parallel for (i, j) ∈ S_U do
4          s = a_ij − Σ_{k=1}^{i−1} u_ki u_kj
5          if i ≠ j then
6              u_ij = s / u_ii
7          else
8              u_ii = √s
9          end
10     end
11 end
```