Lecture 4: Parallel Programming with the Message Passing Interface (MPI) (Part 3)
Collective Communication in MPI

• All or None:
  ➢ Collective communication must involve all processes in the scope of a communicator.
  ➢ It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations
    – must be called by all processes in a communicator

• Types of Collective Operations:
  ➢ Synchronization - processes wait until all members of the group have reached the synchronization point.
  ➢ Data Movement - broadcast, scatter/gather, all to all.
  ➢ Collective Computation (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.
Collective Communication Routines

• Programming considerations and restrictions
  ➢ Collective operations are blocking, but synchronization is not guaranteed (except for BARRIER) in MPI.
  ➢ No tags.
  ➢ Can only be used with MPI predefined datatypes - not with MPI derived data types.
  ➢ Receive buffers must be exactly the right size
Collective Operations

Barrier synchronization:

```c
int MPI_Barrier(MPI_Comm comm);
```
- returns only after all the processes in the group have called this function.

Broadcast:

```c
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
               int source, MPI_Comm comm)
```
- sends the data stored in buffer of ‘source’ to all the other processes in the group.
- all processes must specify same source rank and communicator
Collective Operations

Reduction:

int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm comm)

- combines the elements stored in ‘sendbuf’ of each process in the group, using the operation op, and returns the combined values in the ‘recvbuf’ of the target process.

int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- returns the combined values to all processes.
An example for MPI_Reduce

```c
#include <mpi.h>
void main (int argc, char *argv[]) {
    int rank;

    struct {
        double value;
        int rank;
    } in, out;

    int root;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    in.value=rank+1;
    in.rank=rank;
    root=7;

    MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
    if(rank==root) printf("PE:%d max=%lf at rank %d\n",rank,out.value,out.rank);
    MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MINLOC,root,MPI_COMM_WORLD);
    if(rank==root) printf("PE:%d min=%lf at rank %d\n",rank,out.value,out.rank);
    MPI_Finalize();
}
```
Collective Operations

Prefix operation:

```c
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- the receive buffer of process $i$ will store, at the end of the operation, the reduction of the send buffers of the processes 0 up to and including $i$. 

```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```
Collective Operations

Gather:

```c
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int target, MPI_Comm comm)
```

- each process sends the data stored in `sendbuf` to the target process (target process receives the content of `p` buffers).
Collective Operations

All-Gather:

\[
\text{int } \text{MPI\_Allgather}(\text{void }*\text{sendbuf, int } \text{sendcount, MPI\_Datatype } \text{sendtype,}
\]
\[
\text{void }*\text{recvbuf, int } \text{recvcount, MPI\_Datatype } \text{recvtype, MPI\_Comm } \text{comm});
\]

- the data are gathered to all the processes and not only at the target process.

Vector Variants for Gather: \text{MPI\_Gatherv(), MPI\_Allgatherv()}

- the size of the arrays send by each process can be different.
Collective Operations

**Scatter:**

```c
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int revcnt, MPI_Datatype recvtype, int source, MPI_Comm comm)
```

- the source process sends a different part of the send buffer to each process, including itself.
- process i receives sendcnt elements starting from \(i \times \text{sendcnt}\) location of sendbuf.

Can you use only send()/receive() to implement this function?
Collective Operations

All-to-All:

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)
```

- Each process sends a different part of the send buffer to each other processes, including itself.
  - Each process sends to process i `sendcount` elements starting from `i*sendcount` location of `sendbuf`.
- Each process receives from process i `recvcnt` elements and store them in `recvbuf` starting at location `i*recvcnt`.
- MPI provides also a vector variant.
Collective Operations

All-to-All diagram:

```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
MPI_COMM_WORLD);
```

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

sendbuf (before)

recvbuf (after)
Example: Matrix-Vector Multiplication

(a) Column major data access

(b) Row major data access.
Example: Row-wise Matrix-Vector Multiplication

```c
RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
    MPI_Comm comm)
{
    int i, j;
    int nlocal;  /* Number of locally stored rows of A */
    double *fb;  /* Will point to a buffer that stores the entire vector b */
    int npes, myrank;
    MPI_Status status;

    /* Get information about the communicator */
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    /* Allocate the memory that will store the entire vector b */
    fb = (double *)malloc(n*sizeof(double));

    nlocal = n/npes;

    /* Gather the entire vector b on each processor using MPI's ALLGATHER */
    MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE, comm);

    /* Perform the matrix-vector mult. Involving locally stored submatrix */
    for (i=0; i<nlocal; i++){
        x[i] = 0.0;
        for (j=0; j<n; j++)
            x[i] += a[i*n+j]*fb[j];
    }

    free(fb);
}
```
Example: Column-wise Matrix-Vector Multiplication

```c
ColMatrixVectorMultiply(int n, double *a, double *b, double *x,
                        MPI_Comm comm) {
  int i, j;
  int nlocal;
  double *px;
  double *fx;
  int npes, myrank;
  MPI_Status status;

  /* Get identity and size information from the communicator */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  nlocal = n/npes;

  /* Allocate memory for arrays storing intermediate results. */
  px = (double *)malloc(n*sizeof(double));
  fx = (double *)malloc(n*sizeof(double));

  /* Compute the partial-dot products that correspond to the local columns of A. */
  for (i=0; i<n; i++) {
    px[i] = 0.0;
    for (j=0; j<nlocal; j++)
      px[i] += a[i*nlocal+j]*b[j];
  }

  /* Sum-up the results by performing an element-wise reduction operation */
  MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);

  /* Redistribute fx in a fashion similar to that of vector b */
  MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0, comm);
  free(px); free(fx);
}
```
Example: Dijkstra’s Single-Source Shortest-Paths Algorithm

For a weighted graph $G = (V, E, w)$, the problem is to find the shortest paths from vertex $v \in V$ to all other vertices in $V$. A shortest path from $u$ to $v$ is a minimal path.

---

**Procedure Dijkstra_single_source_SP(V, E, w, s)**

\[ V_T \leftarrow \{s\} \]

\[ \text{for all } v \in V - V_T \text{ do} \]
\[ \text{if } (s, v) \in E \text{ then } \ell[v] \leftarrow w(s, v) \]
\[ \text{else } \ell[v] \leftarrow \infty \]
\[ \text{end for} \]

\[ \text{while } V_T \neq V \text{ do} \]
\[ \text{find } u \in V - V_T \text{ such that } \ell[u] = \min \{ \ell[v] | v \in V - V_T \} \]
\[ V_T \leftarrow V_T \cup \{u\} \]
\[ \text{for all } v \in V - V_T \text{ do} \]
\[ \ell[v] \leftarrow \min \{ \ell[v], \ell[u] + w(u, v) \} \]
\[ \text{end for} \]
\[ \text{end while} \]
Example: Dijkstra’s Single-Source Shortest-Paths Algorithm
Dijkstra’s Algorithm

```c
SingleSource(int n, int source, int *wgt, int *lengths, MPI_Comm comm)
{
  int i, j;
  int nlocal; /* The number of vertices stored locally */
  int *marker; /* Used to mark the vertices belonging to \( V_0 \) */
  int firstvtx; /* The index number of the first vertex that is stored locally */
  int lastvtx; /* The index number of the last vertex that is stored locally */
  int u, udist;
  int lminpair[2], gminpair[2];
  int npes, myrank;
  MPI_Status status;

  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  nlocal = n/npes;
  firstvtx = myrank*nlocal;
  lastvtx = firstvtx+nlocal-1;

  /* Set the initial distances from source to all the other vertices */
  for (j=0; j<nlocal; j++)
    lengths[j] = wgt[source*nlocal + j];

  /* This array is used to indicate if the shortest path to a vertex has been found or not. */
  /* if marker[v] is zero then the shortest path to v has been found, otherwise it has not. */
  marker = (int *)malloc(nlocal*sizeof(int));
  for (j=0; j<nlocal; j++)
    marker[j] = 1;

  /* The process that stores the source vertex, marks it as being seen */
  if (source >= firstvtx && source <= lastvtx)
    marker[source-firstvtx] = 0;
}```
Dijkstra's Algorithm (Cont'd)

/* The main loop of Dijkstra's algorithm */
for (i=1; i<n; i++) {
    /* Step 1: Find the local vertex that is at the smallest distance from source */
    lminpair[0] = MAXINT; /* set it to an architecture dependent large number */
    lminpair[1] = -1;
    for (j=0; j<nlocal; j++) {
        if (marker[j] && lengths[j] < lminpair[0]) {
            lminpair[0] = lengths[j];
            lminpair[1] = firstvtx+j;
        }
    }

    /* Step 2: Compute the global minimum vertex, and insert it into Vc */
    MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC, comm);
    udist = gminpair[0];
    u = gminpair[1];

    /* The process that stores the minimum vertex, marks it as being seen */
    if (u == lminpair[1])
        marker[u-firstvtx] = 0;

    /* Step 3: Update the distances given that u got inserted */
    for (j=0; j<nlocal; j++) {
        if (marker[j] && udist + wgt[u*nlocal+j] < lengths[j],
            lengths[j] = udist + wgt[u*nlocal+j];
    }
}

free(marker);
Example: Samplesort

Initial element distribution

Local sort & sample selection

Sample combining

Global splitter selection

Final element assignment
Example Samplesort (Cont’d)

```c
int *SampleSort(int n, int *elmnts, int *nsorted, MPI_Comm comm)
{
    int i, j, nlocal, npes, myrank;
    int *sorted_elmnts, *splitters, *allpicks;
    int *scounts, *sdispls, *rcounts, *rdispls;

    /* Get communicator-related information */
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    nlocal = n/npes;

    /* Allocate memory for the arrays that will store the splitters */
    splitters = (int *)malloc(npes*sizeof(int));
    allpicks = (int *)malloc(npes*(npes-1)*sizeof(int));

    /* Sort local array */
    qsort(elmnts, nlocal, sizeof(int), IncOrder);

    /* Select local npes-1 equally spaced elements */
    for (i=1; i<npes; i++)
        splitters[i-1] = elmnts[i*nlocal/npes];

    /* Gather the samples in the processors */
    MPI_Allgather(splitters, npes-1, MPI_INT, allpicks, npes-1,
                  MPI_INT, comm);

    /* sort these samples */
    qsort(allpicks, npes*(npes-1), sizeof(int), IncOrder);

    /* Select splitters */
    for (i=1; i<npes; i++)
        splitters[i-1] = allpicks[i*npes];
    splitters[npes-1] = MAXINT;

    /* Compute the number of elements that belong to each bucket */
    scounts = (int *)malloc(npes*sizeof(int));
    for (i=0; i<npes; i++)
        scounts[i] = 0;
}```
Example Samplesort (Cont’d)

```c
for (j=i=0; i<nlocal; i++) {
    if (elmnts[i] < splitters[j])
        scounts[j]++;
    else
        scounts[++j]++;
}

/* Determine the starting location of each bucket's elements in the elmnts array */
sdispls = (int *)malloc(npes*sizeof(int));
sdispls[0] = 0;
for (i=1; i<npes; i++)
    sdispls[i] = sdispls[i-1]+scounts[i-1];

/* Perform an all-to-all to inform the corresponding processes of the number of elements */

/* they are going to receive. This information is stored in rcounts array */
rcounts = (int *)malloc(npes*sizeof(int));
MPI_Alltoall(scounts, 1, MPI_INT, rcounts, 1, MPI_INT, comm);

/* Based on rcounts determine where in the local array the data from each processor */
/* will be stored. This array will store the received elements as well as the final */
/* sorted sequence */
rdispls = (int *)malloc(npes*sizeof(int));
rdispls[0] = 0;
for (i=1; i<npes; i++)
    rdispls[i] = rdispls[i-1]+rcounts[i-1];

*nsorted = rdispls[npes-1]+rcounts[i-1];
sorted_elmnts = (int *)malloc(*nsorted*sizeof(int));
```

Example Samplesort (Cont’d)

/* Each process sends and receives the corresponding elements, using the MPI_Alltoallv */
/* operation. The arrays scounts and sdispls are used to specify the number of elements */
/* to be sent and where these elements are stored, respectively. The arrays rcounts */
/* and rdispls are used to specify the number of elements to be received, and where these */
/* elements will be stored, respectively. */
MPI_Alltoallv(elmnts, scounts, sdispls, MPI_INT, sorted_elmnts,
             rcounts, rdispls, MPI_INT, comm);

/* Perform the final local sort */
qsort(sorted_elmnts, *nsorted, sizeof(int), IncOrder);

free(splitters); free(allpicks); free(scounts); free(sdispls);
free(rcounts); free(rdispls);

return sorted_elmnts;
}
Groups and Communication

Partitioning a graph of processes:

```c
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

- called by all the processes in the communicator.
- a subgroup contains all processes that have the same value for the color parameter.
- within each subgroup processes are ranked using the key parameter.
Splitting Cartesian Topologies

`int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims, MPI_Comm *comm_subcart)`

- partitions a cartesian topology into lower dimensional grids.
- if `keep_dims = true` then the i-th dimension is retained in the new sub-topology.
Example: 2D Matrix-Vector Multiplication

```c
MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,
                         MPI_Comm comm)
{
    int ROW=0, COL=1;  /* Improve readability */
    int i, j, nlocal;
    double *px;      /* Will store partial dot products */
    int npes, dims[2], periods[2], keep_dims[2];
    int myrank, my2drank, mycoords[2];
    int other_rank, coords[2];
    MPI_Status status;
    MPI_Comm comm_2d, comm_row, comm_col;

    /* Get information about the communicator */
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    /* Compute the size of the square grid */
    dims[ROW] = dims[COL] = sqrt(npes);
    nlocal = n/dims[ROW];

    /* Allocate memory for the array that will hold the partial dot-products */
    px = malloc(nlocal*sizeof(double));

    /* Set up the Cartesian topology and get the rank & coordinates of the process in this topology */
    periods[ROW] = periods[COL] = 1;  /* Set the periods for wrap-around connections */
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);
    MPI_Comm_rank(comm_2d, &my2drank);  /* Get my rank in the new topology */
    MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);  /* Get my coordinates */

    /* Create the row-based sub-topology */
    keep_dims[ROW] = 0;
    keep_dims[COL] = 1;
    MPI_Cart_sub(comm_2d, keep_dims, &comm_row);

    /* Create the column-based sub-topology */
    keep_dims[ROW] = 1;
    keep_dims[COL] = 0;
    MPI_Cart_sub(comm_2d, keep_dims, &comm_col);

    /* Redistribute the b vector. */
    /* Step 1. The processors along the 0th column send their data to the diagonal processors */
    if (mycoords[COL] == 0 && mycoords[ROW] != 0) {  /* I'm in the first column */
        coods[ROW] = mycoords[ROW];
        coords[COL] = mycoords[ROW];
        MPI_Cart_rank(comm_2d, coords, &other_rank);
```
Example: 2D Matrix-Vector Multiplication (Cont’d)

```c
MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, Comm_2d)
}
if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0)
  coords[ROW] = mycoords[ROW];
  coords[COL] = 0;
  MPI_Cart_rank(comm_2d, coords, &other_rank);
  MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, Comm_2d)
        &status);
}

/* Step 2. The diagonal processors perform a column-wise broadcast */
coords[0] = mycoords[COL];
MPI_Cart_rank(comm_col, coords, &other_rank);
MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);

/* Get into the main computational loop */
for (i=0; i<nlocal; i++) {
  px[i] = 0.0;
  for (j=0; j<nlocal; j++)
    px[i] += a[i*nlocal+j]*b[j];
}

/* Perform the sum-reduction along the rows to add up the partial dot-products */
coords[0] = 0;
MPI_Cart_rank(comm_row, coords, &other_rank);
MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank,       
  comm_row);

MPI_Comm_free(&comm_2d); /* Free up communicator */
MPI_Comm_free(&comm_row); /* Free up communicator */
MPI_Comm_free(&comm_col); /* Free up communicator */
free(px);
}