Introduction to parallel computing

Distributed Memory Programming with MPI (3)

Zhiao Shi (Modifications by Will French)
Advanced Computing Center for Research & Education
Communication Modes

• **Standard** mode
  • Buffering is system dependent.

• **Buffered** mode
  • A buffer must be provided by the application.

• **Synchronous** mode
  • Completes only after a matching receive has been posted.

• **Ready** mode
  • May only be called when a matching receive has already been posted.
Collective Communications

• Collective communications refer to set of MPI functions that transmit data among all processes specified by a given communicator.
• Three general classes:
  • Barrier
  • Global communication (broadcast, gather, scatter)
  • Global reduction
• Collective functions are less flexible than point-to-point in the following ways:
  • Amount of data sent must exactly match amount of data specified by receiver
  • No tag argument
  • Blocking versions only (until MPI 3.0)
  • Only one mode (until MPI 3.0)
Barrier: MPI_Barrier

- **MPI_Barrier** (MPI_Comm comm)
  - IN : comm (communicator)
- Blocks each calling process until all processes in communicator have executed a call to *MPI_Barrier*.
- Used whenever you need to enforce ordering on the execution of the processors:
  - Expensive operation
Global Operations

- MPI_Bcast
- MPI_Gather
- MPI_Scatter
- MPI_Allgather
- MPI_Alltoall
MPI_Bcast

\[ A_0 \]: any chunk of contiguous data described with MPI_Type and count
MPI_Bcast

• MPI_Bcast (void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm)

    INOUT  : buffer (starting address, as usual)
    IN     : count  (num entries in buffer)
    IN     : type    (can be user-defined)
    IN     : root    (rank of broadcast root)
    IN     : comm    (communicator)

• Broadcasts message from root to all processes (including root). On return, contents of buffer is copied to all processes in comm.
/* includes here */

int main(int argc, char **argv){
    int mype, nprocs;
    float data = -1.0;
    FILE *file;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &mype);

    if (mype == 0){
        char input[100];
        file = fopen("data1.txt", "r");
        assert (file != NULL);
        fscanf(file, "%s\n", input);
        data = atof(input);
    }
    printf("data before: %f\n", data);
    MPI_Bcast(&data, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    printf("data after: %f\n", data);

    MPI_Finalize();
}

Read a parameter file on a single processor and send data to all processes.
MPI\_Gather, MPI\_Scatter

\begin{align*}
\text{data} & \quad \rightarrow \\
A_0 & \quad A_1 \\
A_2 & \quad A_3 \\
A_4 & \quad A_5
\end{align*}

\begin{align*}
\text{processes} & \quad \nabla \quad \nabla \\
\text{data} & \quad \rightarrow \\
A_0 & \quad \nabla \\
A_1 & \quad \nabla \\
A_2 & \quad \nabla \\
A_3 & \quad \nabla \\
A_4 & \quad \nabla \\
A_5 & \quad \nabla
\end{align*}

\text{Scatter} \quad \rightarrow \quad \text{Gather}
MPI_Gather

• **MPI_Gather** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

  • **IN** sendbuf  (starting address of send buffer)
  • **IN** sendcount  (number of elements in send buffer)
  • **IN** sendtype  (type)
  • **OUT** recvbuf  (address of receive bufer)
  • **IN** recvcount  (n-elements for any single receive)
  • **IN** recvtype  (data type of recv buffer elements)
  • **IN** root  (rank of receiving process)
  • **IN** comm  (communicator)
MPI_Gather

• Each process sends content of send buffer to the root process.
• Root receives and stores in rank order.
• Note: Receive buffer argument ignored for all non-root processes (also recvtype, etc.)
• Also, note that recvcount on root indicates number of items received from each process, not total. This is a very common error.
int rank, nproc;
int root = 0;
int *data_received=NULL, data_send[100];

// assume running with 10 cpus
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
if( rank==root )
    data_received = malloc( sizeof(int)*100*nproc ); // 100*10

// each process sets the value of data_send

MPI_Gather(data_send, 100, MPI_INT, data_received, 100, MPI_INT, root, MPI_COMM_WORLD); // ok
// MPI_Gather(data_send,100,MPI_INT,data_received, 100*nproc, MPI_INT, root, MPI_COMM_WORLD); ← wrong!
MPI_Scatter

- MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

  - IN  sendbuf   (starting address of send buffer)
  - IN  sendcount (number of elements sent to each process)
  - IN  sendtype  (type)
  - OUT recvbuf  (address of receive buffer)
  - IN  recvcount (n-elements in receive buffer)
  - IN  recvtype  (data type of receive elements)
  - IN  root      (rank of sending process)
  - IN  comm      (communicator)
MPI_Scatter

• Inverse of MPI_Gather
• Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter.
• **Note:** all arguments are significant on root, while on other processes only recvbuf, recvcount, recvtype, root, and comm are significant.
Example usages

- Scatter: Create a distributed array from a serial one.
- Gather: Create a serial array from a distributed one.
int A[1000], B[100];
... // initialize A etc

// assume 10 processors
MPI_Scatter(A, 100, MPI_INT, B, 100, MPI_INT, 0,
            MPI_COMM_WORLD); // is this ok?
...
MPI_Scatter(A, 1000, MPI_INT, B, 100, MPI_INT, 0,
            MPI_COMM_WORLD); // is this ok?
Scatter Example

```c
int A[1000], B[100];
... // initialize A etc

// assume 10 processors
MPI_Scatter(A, 100, MPI_INT, B, 100, MPI_INT, 0, MPI_COMM_WORLD); // is this ok?

MPI_Scatter(A, 1000, MPI_INT, B, 100, MPI_INT, 0, MPI_COMM_WORLD); // is this ok?
```
MPI_Allgather

data

<table>
<thead>
<tr>
<th>processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
</tr>
<tr>
<td>B₀</td>
</tr>
<tr>
<td>C₀</td>
</tr>
<tr>
<td>D₀</td>
</tr>
<tr>
<td>E₀</td>
</tr>
<tr>
<td>F₀</td>
</tr>
</tbody>
</table>

allgather

data

<table>
<thead>
<tr>
<th>processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
</tr>
<tr>
<td>B₀</td>
</tr>
<tr>
<td>C₀</td>
</tr>
<tr>
<td>D₀</td>
</tr>
<tr>
<td>E₀</td>
</tr>
<tr>
<td>F₀</td>
</tr>
<tr>
<td>A₀</td>
</tr>
<tr>
<td>B₀</td>
</tr>
<tr>
<td>C₀</td>
</tr>
<tr>
<td>D₀</td>
</tr>
<tr>
<td>E₀</td>
</tr>
<tr>
<td>F₀</td>
</tr>
<tr>
<td>A₀</td>
</tr>
<tr>
<td>B₀</td>
</tr>
<tr>
<td>A₀</td>
</tr>
<tr>
<td>D₀</td>
</tr>
<tr>
<td>E₀</td>
</tr>
<tr>
<td>F₀</td>
</tr>
</tbody>
</table>
MPI_Allgather

- MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

  - IN sendbuf (starting address of send buffer)
  - IN sendcount (number of elements in send buffer)
  - IN sendtype (type)
  - OUT recvbuf (address of receive buffer)
  - IN recvcount (n-elements received from any proc)
  - IN recvtype (data type of receive elements)
  - IN comm (communicator)
MPI_Allgather

• Each process has some chunk of data. Collect in a rank-order array on a single proc and broadcast this out to all procs.

• Like MPI_Gather except that all processes receive the result (instead of just root).
MPI_Alltoall

```
<table>
<thead>
<tr>
<th>processes</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aₐ₀</td>
<td>A₁</td>
</tr>
<tr>
<td>B₀</td>
<td>B₁</td>
</tr>
<tr>
<td>C₀</td>
<td>C₁</td>
</tr>
<tr>
<td>D₀</td>
<td>D₁</td>
</tr>
<tr>
<td>E₀</td>
<td>E₁</td>
</tr>
<tr>
<td>F₀</td>
<td>F₁</td>
</tr>
</tbody>
</table>

alltoall

```

```
<table>
<thead>
<tr>
<th>processes</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>B₀</td>
</tr>
<tr>
<td>A₁</td>
<td>B₁</td>
</tr>
<tr>
<td>A₂</td>
<td>B₂</td>
</tr>
<tr>
<td>A₃</td>
<td>B₃</td>
</tr>
<tr>
<td>A₄</td>
<td>B₄</td>
</tr>
<tr>
<td>A₅</td>
<td>B₅</td>
</tr>
</tbody>
</table>
```
MPI_Alltoall

- MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

- IN sendbuf  (starting address of send buffer)
- IN sendcount (number of elements sent to each proc)
- IN sendtype  (type)
- OUT recvbuf  (address of receive buffer)
- IN recvcount (n-elements in receive buffer)
- IN recvtype  (data type of receive elements)
- IN comm      (communicator)

- MPI_Alltoall is an extension of MPI_Allgathere to case where each process sends distinct data to each reciever.
Global reduction operations

- MPI_Reduce
- MPI_Allreduce

<table>
<thead>
<tr>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

reduce

<table>
<thead>
<tr>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

allreduce

<table>
<thead>
<tr>
<th>A0+A1+A2</th>
<th>B0+B1+B2</th>
<th>C0+C1+C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
<tr>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
<tr>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
</tbody>
</table>
MPI_Reduce

- MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - IN sendbuf (address of send buffer)
  - OUT recvbuf (address of receive buffer)
  - IN count (number of elements in send buffer)
  - IN datatype (data type of elements in send buffer)
  - IN op (reduce operation)
  - IN root (rank of root process)
  - IN comm (communicator)
MPI_Reduce

• MPI_Reduce combines elements specified by send buffer and performs a reduction operation on them.
• There are a number of predefined reduction operations: MPI_MAX, MPI_MIN, MPI_SUM, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC
MPI_Allreduce

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

- IN  sendbuf    (address of send buffer)
- OUT recvbuf    (address of receive buffer)
- IN  count      (number of elements in send buffer)
- IN  datatype    (data type of elements in send buffer)
- IN  op          (reduce operation)
- IN  comm        (communicator)
Collective vs. Point-to-Point Communications

• All the processes in the communicator must call the same collective function.
• Point-to-point communications are matched on the basis of tags and communicators.
• Collective communications don’t use tags.
• They’re matched solely on the basis of the communicator and the order in which they’re called.
Next time

• User-defined data type
• Performance measurements