Introduction to Parallel Programming with MPI

Misha Sekachev (mikhail@utk.edu)
Christian Halloy (halloy@jics.utk.edu)

JICS Research and Support
NICS Scientific Computing
Outline: Collective Communications

- **Overview**
- **Barrier Synchronization Routines**
- **Broadcast Routines**
- **MPI_Scatterv and MPI_Gatherv**
- **MPI_Allgather**
- **MPI_Alltoall**
- **Global Reduction Routines**
- **Reduce and Allreduce**
- **Predefined Reduce Operations**
Overview

• Substitutes for a more complex sequence of point-to-point calls
• Involve all the processes in a process group
• Called by all processes in a communicator
• All routines block until they are locally complete
• Receive buffers must be exactly the right size
• No message tags are needed
• Divided into three subsets:
  – synchronization
  – data movement
  – global computation
Barrier Synchronization Routines

- To synchronize all processes within a communicator
- A node calling it will be blocked until all nodes within the group have called it.
- C:
  ```c
  ierr = MPI_Barrier(comm)
  ```
- Fortran:
  ```fortran
  call MPI_Barrier(comm, ierr)
  ```
- C++:
  ```cpp
  void MPI::Comm::Barrier() const;
  ```
Broadcast Routines

• One processor sends some data to all processors in a group

C:

    ierr = MPI_Bcast(buffer, count, datatype, root, comm)

Fortran:

    call MPI_Bcast(buffer, count, datatype, root, comm, ierr)

C++:

    void MPI::Comm::Bcast(void* buffer, int count, const MPI::Datatype& datatype, int root) const;

• The MPI_Bcast must be called by each node in a group, specifying the same communicator and root. The message is sent from the root process to all processes in the group, including the root process.
Scatter

- Data are distributed into \( n \) equal segments, where the \( i^{th} \) segment is sent to the \( i^{th} \) process in the group which has \( n \) processes.

**C:**

```c
ierr = MPI_Scatter(&sbuff, scount, sdatatype, &rbuf, rcount, rdatatype, root, comm)
```

**Fortran:**

```fortran
call MPI_Scatter(sbuff, scount, sdatatype, rbuf, rcount, rdatatype, root, comm, ierr)
```

**C++:**

```cpp
void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int root) const;
```
Example: MPI_Scatter

real sbuf(12), rbuf(2)
call MPI_Scatter(sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 3, MPI_COMM_WORLD, ierr)
Scatter and Gather

<table>
<thead>
<tr>
<th>PE 0</th>
<th>PE 1</th>
<th>PE 2</th>
<th>PE 3</th>
<th>PE 4</th>
<th>PE 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
</tr>
</tbody>
</table>

scatter

<table>
<thead>
<tr>
<th>PE 0</th>
<th>PE 1</th>
<th>PE 2</th>
<th>PE 3</th>
<th>PE 4</th>
<th>PE 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
</tr>
</tbody>
</table>

gather
Gather

- Data are collected into a specified process in the order of process rank, reverse process of scatter.

- C:
  \[
  \text{ierr} = \text{MPI	extunderscore Gather}(&\text{sbuf}, \text{scount}, \text{sdatatype}, &\text{rbuf}, \text{rcount}, \text{rdatatype}, \text{root}, \text{comm})
  \]

- Fortran:
  \[
  \text{call MPI	extunderscore Gather(sbuff, scount, sdatatyp, rbuff, rcount, rdatatyp, root, comm, ierr)}
  \]

- C++:
  \[
  \text{void MPI::Comm::Gather(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int root) const;}
  \]
Example: MPI_Gather

real sbuf(2), rbuf(12)
call MPI_Gather(sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 3, MPI_COMM_WORLD, ierr)
MPI_Scatterv and MPI_Gatherv

- allow varying count of data and flexibility for data placement

- C:
  
  ```c
  ierr = MPI_Scatterv( &sbuf, &scount, &displace, sdatatype, &rbuf, rcount, rdatatype, root, comm)
  ierr = MPI_Gatherv(&sbuf, scount, sdatatype, &rbuf, &rcount, &displace, rdatatype, root, comm)
  ```

- Fortran:
  
  ```fortran
  call MPI_Scatterv(sbuf,scount,displace,sdatatype, rbuf, rcount, rdatatype, root, comm, ierr)
  ```

- C++:
  
  ```cpp
  void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], const MPI::Datatype& sendtype, void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int root) const;
  ```
MPI\_Allgather

\[
ierr = MPI\_Allgather(&sbuf, scount, stype, &rbuf, rcount, rtype, comm)
\]

| PE 0 | A0 | |
| PE 1 | B0 | |
| PE 2 | C0 | |
| PE 3 | D0 | |
| PE 4 | E0 | |
| PE 5 | F0 | |

allgather

| PE 0 | A0 B0 C0 D0 E0 F0 |
| PE 1 | A0 B0 C0 D0 E0 F0 |
| PE 2 | A0 B0 C0 D0 E0 F0 |
| PE 3 | A0 B0 C0 D0 E0 F0 |
| PE 4 | A0 B0 C0 D0 E0 F0 |
| PE 5 | A0 B0 C0 D0 E0 F0 |
MPI_Alltoall

MPI_Alltoall(sbuf, scount, stype, rbuf, rcount, rtype, comm)

sbuf: starting address of send buffer (*)
scount: number of elements sent to each process
stype: data type of send buffer
rbuff: address of receive buffer (*)
rcount: number of elements received from any process
rtype: data type of receive buffer elements
comm: communicator
### All to All

<table>
<thead>
<tr>
<th>PE 0</th>
<th>PE 1</th>
<th>PE 2</th>
<th>PE 3</th>
<th>PE 4</th>
<th>PE 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>B0</td>
<td>C0</td>
<td>D0</td>
<td>E0</td>
<td>F0</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
<td>E1</td>
<td>F1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
<td>E2</td>
<td>F2</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
<td>E3</td>
<td>F3</td>
</tr>
<tr>
<td>A4</td>
<td>B4</td>
<td>C4</td>
<td>D4</td>
<td>E4</td>
<td>F4</td>
</tr>
<tr>
<td>A5</td>
<td>B5</td>
<td>C5</td>
<td>D5</td>
<td>E5</td>
<td>F5</td>
</tr>
</tbody>
</table>

The diagram illustrates the all-to-all communication pattern where each processor (PE) sends data to every other processor. The data elements are labeled from A0 to F5.

**Alltoall**
Global Reduction Routines

- The partial result in each process in the group is combined together using some desired function.
- The operation function passed to a global computation routine is either a predefined MPI function or a user supplied function.
- Examples:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation
Reduce and Allreduce

MPI_Reduce(sbuf, rbuf, count, stype, op, root, comm)
MPI_Allreduce(sbuf, rbuf, count, stype, op, comm)

- **sbuf**: address of send buffer
- **rbuf**: address of receive buffer
- **count**: the number of elements in the send buffer
- **stype**: the datatype of elements of send buffer
- **op**: the reduce operation function, predefined or user-defined
- **root**: the rank of the root process
- **comm**: communicator

**Notes**:
- MPI_Reduce returns results to a single process.
- MPI_Allreduce returns results to all processes in the group.
## Predefined Reduce Operations

<table>
<thead>
<tr>
<th>MPI NAME</th>
<th>FUNCTION</th>
<th>MPI NAME</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>MPI_MAXLOC</td>
<td>Maximum and location</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Bitwise AND</td>
<td>MPI_MINLOC</td>
<td>Minimum and location</td>
</tr>
</tbody>
</table>
Example: MPI Collective Communication Functions

- Collective communication routines are a group of MPI message passing routines to perform one (processor)-to-many (processors) and many-to-one communications.
- The first four columns on the left denote the contents of respective send buffers (e.g., arrays) of four processes. The content of each buffer, shown here as alphabets, is assigned a unique color to identify its origin. For instance, the alphabets in blue indicate that they originated from process 1. The middle column shows the MPI routines with which the send buffers are operated on. The four columns on the right represent the contents of the processes' receive buffers resulting from the MPI operations.
Outline: Derived Datatypes

- **Overview**

- **Datatypes**

- **Defining Datatypes**

- **MPI_Type_vector**

- **MPI_Type_struct**
Overview

• To provide a portable and efficient way of communicating mixed types, or non-contiguous types in a single message
  – Datatypes are built from the basic MPI datatypes
  – MPI datatypes are created at run-time through calls to MPI library
• Steps required
  – construct the datatype : define shapes and handle
  – allocate the datatype : commit types
  – use the datatype : use constructors
  – deallocate the datatype : free space
Datatypes

- Basic datatypes:
  \texttt{MPI\_INT, MPI\_REAL, MPI\_DOUBLE, MPI\_COMPLEX,}
  \texttt{MPI\_LOGICAL, MPI\_CHARACTER, MPI\_BYTE, ...}

- MPI also supports array sections and structures through general
datatypes. A general datatypes is a sequence of basic datatypes and
integer byte displacements. These displacements are taken to be
relative to the buffer that the basic datatype is describing. \implies{}
\textit{typemap}

\[
\text{Datatype} = \{(\text{type0, disp0}), (\text{type1, disp1}), \ldots, (\text{typeN, dispN})\}
\]
Defining Datatypes

MPI_Type_contiguous(count, oldtype, newtype, ierr)
‘count’ copies of ‘oldtype’ are concatenated

MPI_Type_vector(count, buffer, strides, oldtype, newtype, ierr)
‘count’ blocks with ‘blen’ elements of ‘oldtype’ spaced by ‘stride’

MPI_Type_indexed(count, buffer, strides, oldtype, newtype, ierr)
Extension of vector: varying ‘blens’ and ‘strides’

MPI_Type_struct(count, buffer, strides, oldtype, newtype, ierr)
extension of indexed: varying data types allowed
MPI_Type_vector

- It replicates a datatype, taking blocks at fixed offsets.

\[
\text{MPI_Type_vector}(\text{count}, \text{blocklen}, \text{stride}, \text{oldtype}, \text{newtype})
\]

- The new datatype consists of:
  - \text{count}: number of blocks
  - each block is a repetition of \text{blocklen} items of \text{oldtype}
  - the start of successive blocks is offset by \text{stride} items of \text{oldtype}

\[
\text{If count} = 2, \text{stride} = 4, \text{blocklen} = 3, \text{oldtype} = \{(\text{double}, 0), (\text{char}, 8)\}
\]
\[
\text{newtype} = \{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24),
(\text{double}, 32), (\text{char}, 40), (\text{double}, 64),
(\text{char}, 72), (\text{double}, 80), (\text{char}, 88),
(\text{double}, 96), (\text{char}, 104)\}
\]
Example: Datatypes

```c
#include <mpi.h>
{
    float mesh[10][20];
    int dest, tag;
    MPI_Datatype newtype;
    /* Do this once */
    MPI_Type_vector(10, /* # column elements */
                    1,    /* 1 column only */
                    20,   /* skip 20 elements */
                    MPI_FLOAT, /* elements are float */
                    &newtype); /* MPI derived datatype */
    MPI_Type_commit(&newtype); /* Do this for every new message */
    MPI_Send(&mesh[0][19], 1, newtype, dest, tag, MPI_COMM_WORLD);
}
```
MPI_Type_struct

To gather a mix of different datatypes scattered at many locations in space into one datatype

MPI_Type_struct(count, array_of_blocklength, array_of_displacements, array_of_types, newtype, ierr)

- count: number of blocks
- array_of_blocklength (B): number of elements in each block
- array_of_displacements (I): byte of displacement of each block
- array_of_type (T): type of elements in each block

If count = 3
T = {MPI_FLOAT, type1, MPI_CHAR}
I = {0,16,26}
B = {2,1,3}
type1 = {(double,0),(char,8)}
newtype = {(float,0),(float,4),(double,16),(char,24),
(char,26),(char,27),(char,28)}
Example

```
Struct{
    char display[50];
    int maxiter;
    double xmin, ymin, xmax, ymax;
    int width, height;
} cmdline;

/* set up 4 blocks */
int blockcounts[4] = {50, 1, 4, 2};
MPI_Datatype types[4];
MPI_Aint displs[4];
MPI_Datatype cmdtype;

/* initialize types and displacements with addresses of items */

MPI_Address(&cmdline.display, &displs[0];
MPI_Address(&cmdline.maxiter, &displs[1];
MPI_Address(&cmdline.xmin, &displs[2];
MPI_Address(&cmdline.width, &displs[3];
types[0] = MPI_CHAR; types[1]=MPI_INT;

for ( i = 3 ; i >= 0; i-- )  displs[i] -= displs[0]
MPI_Type_struct(4, blockcounts, displs, types, &cmdtype);
MPI_Type_commit(&cmdtype);
```
example

Struct{  char  display[50];
  int    maxiter;
  double xmin,ymin, xmax, ymax;
  int    width, height;
} cmdline;

/* set up 4 blocks */
Int      blockcounts[4] = {50, 1, 4, 2};
MPI_Datatype types[4];
MPI_Aint  displs[4];
MPI_Datatype cmdtype;

/* initialize types and displacements with addresses of items */
MPI_Address(&cmdline.display, &displs[0]);
MPI_Address(&cmdline.maxiter, &displs[1]);
MPI_Address(&cmdline.xmin, &displs[2]);
MPI_Address(&cmdline.width, &displs[3]);
For ( I =3 ; I >= 0; i-- ) displs[i] - = displs[0]
MPI_Type_struct(4, blockcounts, displs, types, &cmdtype);
MPI_Type_commit(&cmdtype);
Allocate the datatype

- A constructed datatype must be committed to the system before it can be used for communication.
- `MPI_Type_commit(newtype)`

```fortran
integer type1, type2, ierr
call MPI_Type_contiguous( 5, MPI_REAL, type1, ierr)
call MPI_Type_commit( type1, ierr)
type2 = type1
call MPI_Type_vector(3, 5, 4, MPI_REAL, type1, ierr)
call MPI_Type_commit(type1, ierr)
```
Example : derived datatypes

real a(100,100), b(100,100)
integer disp(2), blocklen(2), type(2), row, row1, sizeofreal
integer status(MPI_STATUS_SIZE)
call MPI_Comm_status(MPI_COMM_WORLD, myrank)
call MPI_TypeExtent(MPI_REAL, sizeofreal, ierr)
c transpose matrix a onto b
call MPI_Type_vector(100, 1, 100, MPI_REAL, row, ierr)
c create datatype for one row, with the extent of one real number
disp(1) = 0
disp(2) = sizeofreal
type(1) = row
type(2) = MPI_UB
blocklen(1) = 1
blocklen(2) = 1
call MPI_Type_struct(2, blocklen, disp, type, row1, ierr)
call MPI_Type_comit(row1, ierr)
c send 100 rows and receive in column major order
call MPI_Sendrecv( a, 100, row1, myrank, 0, b, 100*100, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
MPI Group

• To limit communication to a subset of processes, the programmer can create a group, and associate a communicator with that group.

• A group is an ordered set of processes. Each process in a group is associated with a unique integer rank (id). Rank values start at zero and go to N-1, where N is the number of processes in the group.

• New groups or communicators must be created from existing ones.

• Communicator creation routines are collective. They require all processes in the input communicator to participate.
Group Creation

- Access the base group of all processes via a call to MPI_COMM_GROUP
- Create the new group via a call to MPI_GROUP_INCL
- Create the communicator for the new group via a call to MPI_COMM_CREATE

----------------------
example
----------------------

program set_group
include 'mpi.f'
parameter (NROW=2, NCOL=3)
integer row_list(NCOL), base_grp, grp
integer temp_comm, row1_comm, row2_comm
call MPI_Init(ierr)
c----get base group from MPI_COMM_WORLD communicator
call MPI_COMM_GROUP(MPI_COMM_WORLD, base_grp, ierr)
Group Creation (cnt’d)

c----Establish the row to which this processor belongs-----------------
call MPI_COMM_RANK(MPI_COMM_WORLD, irank, ierr)
irow = mod(irank, NROW) + 1
c----build row groups ------------------------------------------------
row_list(1) = 0
do i = 2, NCOL
   row_list(i) = row_list(i-1) + 1
endo
do i = 1, NROW
call MPI_Group_incl(base_grp, NCOL, row_list, grp, ierr)
call MPI_Comm_create(MPI_COMM_WORLD, grp, temp_comm, ierr)
if ( i .eq. 1) row1_comm = temp_comm
if (I .eq. 2) row2_comm = temp_comm
do  j = 1, NCOL
   row_list(j) = row_list(j) + NROW*i +1
endo
doi do
end

call MPI_Finalize(ierr)
end
Group Creation

- MPI_Group_incl (oldgroup, n, ranks, newgroup)
- MPI_Group_excl(oldgroup, n, ranks, newgroup)

main( int argc, char**argv)
{
    MPI_Comm subcomm;
    MPI_Group world_group, subgroup;
    int ranks[]={2,4,6,8}, numprocs,myid
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs)
    MPI_Comm_rank(MPI_COMM_WORLD,&myid)
    MPI_Comm_group(MPI_COMM_WORLD,&world_group);
    MPI_Group_incl(world_group, 4, ranks, &subgroup);
    MPI_Comm_create(MPI_COMM_WORLD, subgroup, &subcomm);
    MPI_Finalize();
}
Group Creation

• An alternate approach is to use MPI_COMM_SPLIT, which partitions one communicator into multiple, non-overlapping communicators.

```fortran
subroutine set_group(row_comm)
  include ‘mpif.h’
  parameter (NROW=2)
  integer row_comm, color, key

  c----Establish the new row to which this processor belongs
  call MPI_COMM_RANK(MPI_COMM_WORLD, irank, ierr)
  irow = mod ( irank, NROW) + 1

  c----build row communicators
  color = irow
  key = irank
  call MPI_COMM_SPLIT(MPI_COMM_WORLD, color, key, row_comm, ierr)
  return
end
```
There are man pages available for MPI which should be installed in your MANPATH. The following man pages have some introductory information about MPI.

- `man MPI`
- `man cc`
- `man ftn`
- `man qsub`
- `man MPI_Init`
- `man MPI_Finalize`

MPI man pages are also available online.

- [Main MPI web page at Argonne National Laboratory](http://www-unix.mcs.anl.gov/mpi)
- [Set of guided exercises](http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl)
- [MPI tutorial at Lawrence Livermore National Laboratory](https://computing.llnl.gov/tutorials/mpi/)
- [MPI Forum home page contains the official copies of the MPI standard.](http://www.mpi-forum.org/)
Resources for Users: MPI Books

- Books on and about MPI
  
  - Designing and Building Parallel Programs is Ian Foster's online book that includes a chapter on MPI. It provides a succinct introduction to an MPI subset. (ISBN 0-201-57594-9; Published by Addison-Wesley)
  
  
  
  - Parallel Programming With MPI, by Peter S. Pacheco, published by Morgan Kaufmann.
  
  - **RS/6000 SP: Practical MPI Programming**, by Yukiya Aoyama and Jun Nakano (IBM Japan), and available as an IBM Redbook.
  