MPI

THOAI NAM
Outline

- Communication modes
- MPI – Message Passing Interface Standard
TERMs (1)

- **Blocking**
  If return from the procedure indicates the user is allowed to reuse resources specified in the call

- **Non-blocking**
  If the procedure may return before the operation completes, and before the user is allowed to reuse resources specified in the call

- **Collective**
  If all processes in a process group need to invoke the procedure

- **Message envelope**
  Information used to distinguish messages and selectively receive them
  
  `<source, destination, tag, communicator>`
TERMs (2)

- **Communicator**
  - The communication context for a communication operation
  - Messages are always received within the context they were sent
  - Messages sent in different contexts do not interfere
  - MPI_COMM_WORLD

- **Process group**
  - The communicator specifies the set of processes that share this communication context.
  - This process group is ordered and processes are identified by their rank within this group
MPI

- Environment
- Point-to-point communication
- Collective communication
- Derived data type
- Group management
MPI

Daemon

P₀

P₁

P₂

P₃

P₄
Environment

- MPI_INIT
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_FINALIZE
- MPI_ABORT
MPI_Init

- **Usage**
  - int MPI_Init( int* argc_ptr, char** argv_ptr[] ); /* in */

- **Description**
  - Initialize MPI
  - All MPI programs must call this routines once and only once before any other MPI routines
MPI_Finalize

- **Usage**
  
  ```c
  int MPI_Finalize (void);
  ```

- **Description**
  
  - Terminates all MPI processing
  - Make sure this routine is the last MPI call.
  - All pending communications involving a process have completed before the process calls MPI_FINALIZE
MPI_Comm_Size

- Usage
  
  int MPI_Comm_size( MPI_Comm comm, /* in */
  int* size ); /* out */

- Description
  
  - Return the number of processes in the group associated with a communicator
MPI_Comm_Rank

- **Usage**
  - `int MPI_Comm_rank ( MPI_Comm comm, /* in */ int* rank ); /* out */`

- **Description**
  - Returns the rank of the local process in the group associated with a communicator.
  - The rank of the process that calls it in the range from 0 … size - 1.
MPI_Abort

- Usage
  - int MPI_Abort( MPI_Comm comm, /* in */
                  int errorcode ); /* in */

- Description
  - Forces all processes of an MPI job to terminate
Simple Program

```
#include "mpi.h"

int main( int argc, char* argv[] )
{
    int rank;
    int nproc;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    /* write codes for you */
    MPI_Finalize();
}
```
Point-to-Point Communication

- MPI_SEND
- MPI_RECV
- MPI_ISEND
- MPI_IRECV
- MPI_WAIT
- MPI_GET_COUNT
Communication Modes in MPI (1)

- **Standard mode**
  - It is up to MPI to decide whether outgoing messages will be buffered
  - Non-local operation
  - Buffered or synchronous?

- **Buffered(asynchronous) mode**
  - A send operation can be started whether or not a matching receive has been posted
  - It may complete before a matching receive is posted
  - Local operation
Synchronous mode

- A send operation can be started whether or not a matching receive was posted.
- The send will complete successfully only if a matching receive was posted and the receive operation has started to receive the message.
- The completion of a synchronous send not only indicates that the send buffer can be reused but also indicates that the receiver has reached a certain point in its execution.
- Non-local operation.
Ready mode

- A send operation may be started only if the matching receive is already posted.
- The completion of the send operation does not depend on the status of a matching receive and merely indicates the send buffer can be reused.
- `EAGER_LIMIT` of SP system.
### MPI_Send

#### Usage

```c
int MPI_Send( void* buf, /* in */
int count,            /* in */
MPI_Datatype datatype, /* in */
int dest,             /* in */
int tag,              /* in */
MPI_Comm comm );      /* in */
```

#### Description

- Performs a blocking standard mode send operation
- The message can be received by either MPI_RECV or MPI_IRECV
MPI_Recv

- **Usage**
  ```c
  int MPI_Recv( void* buf, /* out */
                int count, /* in */
                MPI_Datatype datatype, /* in */
                int source, /* in */
                int tag, /* in */
                MPI_Comm comm, /* in */
                MPI_Status* status ); /* out */
  ```

- **Description**
  - Performs a blocking receive operation
  - The message received must be less than or equal to the length of the receive buffer
  - MPI_RECV can receive a message sent by either MPI_SEND or MPI_ISEND
Process 0

**User Mode**

```
... Call mpi_send(sendbuf, dest=1)  
   (blocked)  
   Now sendbuf can be reused  
...
```

**Kernel Mode**

```
Send data from sysbuf to dest
```

```
Copying data from sendbuf to sysbuf
```

Process 1

**User Mode**

```
... Call mpi_recv(recvbuf, src=0)  
   (blocked)  
   Now recvbuf contains valid data  
...
```

**Kernel Mode**

```
Receive data from src to sysbuf
```

```
Copying data from sysbuf to recvbuf
```

Sample Program for Blocking Operations (1)

#include "mpi.h"

int main( int argc, char* argv[] )
{
    int rank, nproc;
    int isbuf, irbuf;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
if(rank == 0) {
    isbuf = 9;
    MPI_Send( &isbuf, 1, MPI_INTEGER, 1, TAG,
              MPI_COMM_WORLD);
} else if(rank == 1) {
    MPI_Recv( &irbuf, 1, MPI_INTEGER, 0, TAG,
               MPI_COMM_WORLD, &status);
    printf( "%d\n", irbuf );
}

MPI_Finalize();
MPI_Isend

- **Usage**

  ```c
  int MPI_Isend( void* buf, /* in */
               int count, /* in */
               MPI_Datatype datatype, /* in */
               int dest, /* in */
               int tag, /* in */
               MPI_Comm comm, /* in */
               MPI_Request* request ); /* out */
  ```

- **Description**
  - Performs a nonblocking standard mode send operation
  - The send buffer may not be modified until the request has been completed by MPI_WAIT or MPI_TEST
  - The message can be received by either MPI_RECV or MPI_Irecv.
 MPI_Irecv (1)

Usage

```c
int MPI_Irecv( void* buf, /* out */
    int count, /* in */
    MPI_Datatype datatype, /* in */
    int source, /* in */
    int tag, /* in */
    MPI_Comm comm, /* in */
    MPI_Request* request ); /* out */
```
MPI_Irecv (2)

- Description
  - Performs a nonblocking receive operation
  - Do not access any part of the receive buffer until the receive is complete
  - The message received must be less than or equal to the length of the receive buffer
  - MPI_Irecv can receive a message sent by either MPI_SEND or MPI_ISEND
**MPI_Wait**

- **Usage**
  - `int MPI_Wait( MPI_Request* request, /* inout */ MPI_Status* status ); /* out */`

- **Description**
  - Waits for a nonblocking operation to complete
  - Information on the completed operation is found in status.
  - If wildcards were used by the receive for either the source or tag, the actual source and tag can be retrieved by `status->MPI_SOURCE` and `status->MPI_TAG`
**Process 0**

**User Mode**

- Call `mpi_isend(sendbuf, dest, req)`
- (not blocked)
- Call `mpi_wait(req)`
- (blocked)
- Now `sendbuf` can be reused

**Kernel Mode**

- Copying data from `sendbuf` to `sysbuf`
- Send data from `sysbuf` to dest

**Process 1**

**User Mode**

- Call `mpi irecv(recvbuf, src, req)`
- (not blocked)
- Call `mpi_wait(req)`
- (blocked)
- Now `recvbuf` contains valid data

**Kernel Mode**

- Receive data from `src` to `sysbuf`
- Copying data from `sysbuf` to `recvbuf`
MPI_Get_count

- **Usage**
  - int MPI_Get_count( MPI_Status status, /* in */
  MPI_Datatype datatype, /* in */
  int* count ); /* out */

- **Description**
  - Returns the number of elements in a message
  - The datatype argument and the argument provided by the call that set the status variable should match
Sample Program for Non-Blocking Operations (1)

```c
#include "mpi.h"

int main( int argc, char* argv[] )
{
    int rank, nproc;
    int isbuf, irbuf, count;
    MPI_Request request;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if(rank == 0) {
        isbuf = 9;
        MPI_Isend( &isbuf, 1, MPI_INTEGER, 1, TAG, MPI_COMM_WORLD,
                   &request );
    }
}```
} else if (rank == 1) {
    MPI_Irecv( &irbuf, 1, MPI_INTEGER, 0, TAG,
               MPI_COMM_WORLD, &request);
    MPI_Wait(&request, &status);
    MPI_Get_count(&status, MPI_INTEGER, &count);
    printf( "irbuf = %d source = %d tag = %d count = %d\n",
            irbuf, status.MPI_SOURCE, status.MPI_TAG, count);
}

MPI_Finalize();

}
Collective Operations

- MPI_BCAST
- MPI_SCATTER
- MPI_SCATTERV
- MPI_GATHER
- MPI_GATHERV
- MPI_ALLGATHER
- MPI_ALLGATHERV
- MPI_ALLTOALL
MPI_Bcast (1)

- **Usage**
  - int MPI_Bcast( void* buffer, /* inout */
    int count, /* in */
    MPI_Datatype datatype, /* in */
    int root, /* in */
    MPI_Comm comm); /* in */

- **Description**
  - Broadcasts a message from root to all processes in communicator
  - The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root
MPI_Scatter

Usage

```c
int MPI_Scatter( void* sendbuf, /* in */
                int sendcount, /* in */
                MPI_Datatype sendtype, /* in */
                void* recvbuf, /* out */
                int recvcount, /* in */
                MPI_Datatype recvtype, /* in */
                int root, /* in */
                MPI_Comm comm); /* in */
```

Description

- Distribute individual messages from root to each process in communicator
- Inverse operation to MPI_GATHER
Example of MPI_Scatter (1)

#include “mpi.h”

int main( int argc, char* argv[] )
{
    int i;
    int rank, nproc;
    int isend[3], irecv;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
if(rank == 0) {
    for(i=0; i<nproc; i++)
        isend(i) = i+1;
}

MPI_Scatter( isend, 1, MPI_INTEGER, irecv, 1,
            MPI_INTEGER, 0, MPI_COMM_WORLD);

printf("irecv = %d\n", irecv);

MPI_Finalize();
}
Example of MPI_Scatter (3)
## MPI_Scatterv

### Usage

```c
int MPI_Scatterv( void* sendbuf, /* in */
    int* sendcounts, /* in */
    int* displs, /* in */
    MPI_Datatype sendtype, /* in */
    void* recvbuf, /* in */
    int recvcount, /* in */
    MPI_Datatype recvtype, /* in */
    int root, /* in */
    MPI_Comm comm); /* in */
```

### Description

- Distributes individual messages from root to each process in communicator
- Messages can have different sizes and displacements
Example of MPI_Scatterv(1)

```c
#include "mpi.h"

int main( int argc, char* argv[] )
{
    int i;
    int rank, nproc;
    int iscnt[3] = {1,2,3}, irdisp[3] = {0,1,3};
    int isend[6] = {1,2,2,3,3,3}, irecv[3];

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```
Example of MPI_Scatterv(2)

ircnt = rank + 1;

MPI_Scatterv(isend, iscnt, idisp, MPI_INTEGER, irecv, ircnt, MPI_INTEGER, 0, MPI_COMM_WORLD);
printf("irecv = %d\n", irecv);

MPI_Finalize();
}
MPI_Gather

- **Usage**

```c
int MPI_Gather( void* sendbuf, /* in */
                int sendcount, /* in */
                MPI_Datatype sendtype, /* in */
                void* recvbuf, /* out */
                int recvcount, /* in */
                MPI_Datatype recvtype, /* in */
                int root, /* in */
                MPI_Comm comm ); /* in */
```

- **Description**
  - Collects individual messages from each process in communicator to the root process and store them in rank order
Example of MPI_Gather (1)

```c
#include "mpi.h"

int main( int argc, char* argv[] )
{
    int i;
    int rank, nproc;
    int isend, irecv[3];

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```
Example of MPI_Gather (2)

```c
isend = rank + 1;
MPI_Gather( &isend, 1, MPI_INTEGER, irecv, 1,
            MPI_INTEGER, 0, MPI_COMM_WORLD);

if(rank == 0) {
    for(i=0; i<3; i++)
        printf("irecv = %d\n", irecv[i]);

    MPI_Finalize();
}
```
MPI_Gather
Usage

```c
int MPI_Gatherv( void* sendbuf, /* in */
    int sendcount, /* in */
    MPI_Datatype sendtype, /* in */
    void* recvbuf, /* out */
    int* recvcount, /* in */
    int* displs, /* in */
    MPI_Datatype recvtype, /* in */
    int root, /* in */
    MPI_Comm comm ); /* in */
```

Description

- Collects individual messages from each process in communicator to the root process and store them in rank order
Example of MPI_Gatherv (1)

#include "mpi.h"

int main( int argc, char* argv[] )
{
    int i;
    int rank, nproc;
    int isend[3], irecv[6];
    int ircnt[3] = {1,2,3}, idisp[3] = {0,1,3};

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
Example of MPI_Gatherv (2)

```c
for(i=0; i<rank; i++)
    isend[i] = rank + 1;
iscnt = rank + 1;
MPI_Gatherv( isend, iscnt, MPI_INTEGER, irecv, ircnt,
             idisp, MPI_INTEGER, 0, MPI_COMM_WORLD);
if(rank == 0) {
    for(i=0; i<6; i++)
        printf("irecv = %d\n", irecv[i]);
}
MPI_Finalize();
```
MPI_Reduce (1)

- **Usage**

  ```c
  int MPI_Reduce( void* sendbuf, /* in */
                  void* recvbuf, /* out */
                  int count, /* in */
                  MPI_Datatype datatype, /* in */
                  MPI_Op op, /* in */
                  int root, /* in */
                  MPI_Comm comm); /* in */
  ```
Description

- Applies a reduction operation to the vector sendbuf over the set of processes specified by communicator and places the result in recvbuf on root
- Both the input and output buffers have the same number of elements with the same type
- Users may define their own operations or use the predefined operations provided by MPI

Predefined operations

- MPI_SUM, MPI_PROD
- MPI_MAX, MPI_MIN
- MPI_MAXLOC, MPI_MINLOC
- MPI_LAND, MPI_LOR, MPI_LXOR
- MPI_BAND, MPI_BOR, MPI_BXOR
Example of MPI_Reduce

```
#include "mpi.h"
int main( int argc, char* argv[] )
{
    int rank, nproc;
    int isend, irecv;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    isend = rank + 1;
    MPI_Reduce(&isend, &irecv, 1, MPI_INTEGER, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if(rank == 0) printf("irecv = %d\n", irecv);
    MPI_Finalize();
}
```
MPI_Reduce

```
rank=0=root

count
1
sendbuf
+
op
6 = 1 + 2 + 3
recvbuf

rank=1

count
2
sendbuf
+
op

rank=2

count
3
sendbuf
```

comm
MPI_Reduce
MPI_Scan

- **Usage**
  ```c
  int MPI_Scan( void* sendbuf, /* in */
               void* recvbuf, /* out */
               int count, /* in */
               MPI_Datatype datatype, /* in */
               MPI_Op op, /* in */
               MPI_Comm comm); /* in */
  ```

- **Description**
  - Performs a parallel prefix reduction on data distributed across a group
  - The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0…i
Example of MPI_Scan

```c
#include "mpi.h"
int main( int argc, char* argv[] )
{
    int rank, nproc;
    int isend, irecv;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    isend = rank + 1;
    MPI_Scan(&isend, &irecv, 1, MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD);
    printf("irecv = %d\n", irecv);
    MPI_Finalize();
}
```
MPI_Scan

Diagram showing the process of a scan operation in MPI. The diagram illustrates the communication and calculation steps among ranks 0, 1, and 2. Each rank communicates its data to the next rank, and the operation is performed on the received data. The final results are accumulated in the recvbuf.
Usage

```c
int MPI_Barrier(MPI_Comm comm); /* in */
```

Description

- Blocks each process in communicator until all processes have called it