Outline

• Background
  ➢ What is MPI?
  ➢ Why use MPI?

• Basics of MPI message passing
  ➢ What does a MPI program look like?
  ➢ MPI Point-to-Point Communication Routines (blocking vs non-blocking)

• Advanced MPI topics
  ➢ More on MPI datatypes
  ➢ Application topologies
  ➢ Collective operations
What is MPI?

- **MPI** = **Message Passing Interface**
- MPI is a message-passing library specification
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- **MPICH** is a freely available, portable implementation of MPI.
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers
Review of Message-Passing Programming

• Assumes a partitioned address space.
• Supports only explicit parallelism (programmer is responsible for extracting concurrency).
• Logical view: \(p\) processes, each with its own exclusive address space.
• All interaction between processes require cooperation of two processes (sender & receiver).
• Advantages:
  ➢ The programmer is fully aware of the cost of interactions => can minimize them.
  ➢ Can be efficiently implemented on a wide variety of architectures.
• Disadvantage:
  ➢ programming tends to be hard and intellectually demanding.
The Message-Passing Model in MPI

- A process is (traditionally) a program counter and address space.
- MPI is for communication among processes, which have separate address spaces.
  - A process may have multiple threads (program counters and associated stacks) sharing a single address space.
- All parallelism is explicit
  - The programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process’s address space to another’s.
History and Evolution of MPI

• **1980s - early 1990s:** a number of incompatible software tools for writing parallel programs - usually with tradeoffs between portability, performance, functionality and price. Recognition of the need for a standard arose.

• **April, 1992:** a working group established to continue the standardization process.

• **November 1993:** draft MPI standard presented.

• **May 1994:** Final version of draft released.

• **1996:** MPI-2 picked up where the first MPI specification (MPI-1) left off.
Why Use MPI?

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.

- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.

- **Functionality** - Over 115 routines are defined in MPI-1 alone.

- **Availability** - A variety of implementations are available, both vendor and public domain.
Hardware Platforms that Support MPI

- Almost all parallel computers, clusters, and heterogeneous networks

  ➢ Distributed Memory:
    - Originally, MPI was targeted for distributed memory systems.

  ➢ Shared Memory
    - As shared memory systems became more popular, particularly SMP / NUMA architectures, MPI implementations for these platforms appeared.

  ➢ Hybrid:
    - MPI is now used on just about any common parallel architecture including massively parallel machines, SMP clusters, workstation clusters and heterogeneous networks.
What does a MPI program look like?

- Prior to using any MPI functions all processes must call MPI_Init.
  ```c
  int MPI_Init(int *argc, char ***argv)
  ```
- The last MPI function called in all processes must be MPI_Finalize.
  ```c
  int MPI_Finalize()
  ```
- `mpi.h` header file must be included in each MPI program.

```c
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
  MPI_Init( &argc, &argv );
  printf( "Hello, world!\n" );
  MPI_Finalize();
  return 0;
}
```
Format of MPI Routines

- All routines are prefixed by "MPI_" and case-sensitive in C
- By default, an error causes all processes to abort.
Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?

- Communicators and groups: define which collection of processes may communicate with each other.
  - Processes can be collected into groups.
  - Each message is sent in a context, and must be received in the same context.
  - A group and context together form a communicator.
  - Most MPI routines require you to specify a communicator as an argument.
  - The default communicator whose group contains all initial processes is MPI_COMM_WORLD.
Finding Out about the Environment (Cont’d)

• A process is identified by its *rank* in the group associated with a communicator.
  
  - Assigned by the system when the process initializes. Ranks are contiguous and begin at zero.
  
  - Used by the programmer to specify the source and destination of messages.
    
    - Often used conditionally by the application to control program execution (if rank==0 do this / if rank==1 do that).

• MPI provides functions to answer these questions:

  ```c
  int MPI_Comm_size(MPI_Comm comm, int *size)
  int MPI_Comm_rank(MPI_Comm comm, int *rank)
  ```
A Better “Hello! World.”

#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] ) {
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
 MPI Point-to-Point Communication Routines

• Point-to-point operations: involve message passing between two, and only two, different MPI tasks.
  ➢ One task is performing a send operation and the other task is performing a matching receive operation.
• Types of MPI Point-to-Point Operations:
  ➢ Synchronous send
  ➢ Blocking send / blocking receive
  ➢ Non-blocking send / non-blocking receive
• Any type of send routine can be paired with any type of receive routine.
• Other routines that are used to wait for a message's arrival or probe to find out if a message has arrived.
MPI Buffering

• The MPI implementation (not the MPI standard) decides what happens to data in these cases:
  ➢ A send operation occurs 5 seconds before the receive is ready.
  ➢ Multiple sends arrive at the same receiving process which can only accept one send at a time.

• Typically, a system buffer area is reserved to hold data in transit.

• System buffer space is:
  ➢ Opaque to the programmer and managed entirely by the MPI library
  ➢ A finite resource that can be easily exhausted
  ➢ Often mysterious and not well documented
  ➢ Able to exist on the sending side, the receiving side, or both
**Blocking vs. Non-blocking:**

**Blocking:**

- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse.
  - Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive process to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.

---

For synchronous blocking:

(a) Sender comes first; idling at sender
(b) Sender and receiver come at about the same time; idling minimized
(c) Receiver comes first; idling at receiver
• Non-blocking:
  ➢ Non-blocking send and receive return almost immediately.
    – They do not wait for any communication events to complete, such as message
      copying from user memory to system buffer space or the actual arrival of message.
    – Non-blocking operations simply "request" the MPI library to perform the
      operation when it is able. The user cannot predict when that will happen.
    – It is unsafe to modify the application buffer (your variable space) until you know
      for sure the requested non-blocking operation was actually performed by the
      library. There are "wait" routines used to do this.
    – Non-blocking communications are primarily used to overlap computation with
      communication and exploit possible performance gains.

![Diagram of blocking vs. non-blocking operations]
Arguments in MPI Point-to-Point Communication Routines

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking sends</td>
<td><code>MPI_Send(buffer, count, type, dest, tag, comm)</code></td>
</tr>
<tr>
<td>Non-blocking sends</td>
<td><code>MPI_Isend(buffer, count, type, dest, tag, comm, request)</code></td>
</tr>
<tr>
<td>Blocking receive</td>
<td><code>MPI_Recv(buffer, count, type, source, tag, comm, status)</code></td>
</tr>
<tr>
<td>Non-blocking receive</td>
<td><code>MPI_Irecv(buffer, count, type, source, tag, comm, request)</code></td>
</tr>
</tbody>
</table>

- **Buffer**: Process address space that references the data that is to be sent or received. This argument is passed by reference in C.
- **Data Count**: Indicates the number of data elements of a particular type to be sent.
- **Data Type**: MPI predefined elementary data types or user derived data types. Such as `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`
- **Destination**: Indicates the process where a message should be delivered. Specified as the rank of the receiving process.
- **Source**: Indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card `MPI_ANY_SOURCE` to receive a message from any task.
Arguments of MPI Point-to-Point Communication Routines (cont’d)

- **Tag:** Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card `MPI_ANY_TAG` can be used to receive any message regardless of its tag.

- **Communicator:** Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless new communicators are explicitly created, the predefined communicator `MPI_COMM_WORLD` is usually used.

- **Status:** For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure `MPI_Status` (e.g., `stat.MPI_SOURCE stat.MPI_TAG`). Additionally, the actual number of bytes received are obtainable from Status via the `MPI_Get_count` routine.

- **Request:** Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation. In C, this argument is a pointer to a predefined structure `MPI_Request`. 
MPI Blocking Point-to-Point Communication Routines

**MPI_Send (&buf,count, datatype, dest, tag, comm)**
Basic blocking send operation. Whether it is synchronous depends on MPI implementation.

**MPI_Recv (&buf, count, datatype, source, tag, comm, &status)**
Basic blocking receive operation.

**MPI_Ssend (&buf, count, datatype, dest, tag, comm)**
Synchronous blocking send.

**MPI_Sendrecv (&sendbuf, sendcount, sendtype, dest, sendtag, &recvbuf, recvcount,recvtype,source,recvtag, comm, &status)**
Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message. This routine does not suffer from the circular deadlock problems of MPI_Send/Recv.

**MPI_Sendrecv_replace(&buf, count, datatype, dest, sendtag, source, recvtag, comm, &status)**

*(Design experiment to investigate (1) whether the send-then-receive order is enforced in MPI_Sendrecv_replace() and have the risk of deadlock; If not, whether it is possible that the data you send is actually the data you just received. (2) whether it can talk with individual send() and recv())*
Retrieving Further Information

*Status* is a data structure allocated in the user’s program.

```c
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status)
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);
```
A Example using Blocking Routines

```c
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[]; { 
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';
  MPI_Status Stat;

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

  if (rank == 0) {
    dest = 1;
    source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

  else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

  MPI_Finalize();
}
Non-Blocking Communication Operations

• MPI functions for non-blocking send and receive:

  int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
                int dest, int tag, MPI_Comm comm, MPI_Request *request);

  - starts a send operation and returns before the data is copied out of the buffer.

  int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
                int source, int tag, MPI_Comm comm, MPI_Request *request);

  - starts a receive operation and returns before the data has been received and copied into the buffer.

• A non-blocking operation can be matched with a corresponding blocking operation.
Non-Blocking Communication Operations

- MPI functions for checking the completion of non-blocking send and receive:

  ```c
  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
  ```
  - tests whether or not the non-blocking send and receive identified by its request has finished.
  - returns flag = true when operation completed.

  ```c
  int MPI_Wait ( MPI_Request *request, MPI_Status *status)
  ```
  - blocks until the non-blocking operation identified by request completes.

  ```c
  int MPI_Probe (int source, int tag, MPI_Comm comm,
  MPI_Status *status)
  ```
  - Performs a blocking test (wait) for a message to arrive. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag.
A Example using Non-blocking Routines

```c
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[]; 
{
    int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
    MPI_Request reqs[4];
    MPI_Status stats[4];
    
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    
    prev = rank-1;
    next = rank+1;
    if (rank == 0) prev = numtasks - 1;
    if (rank == (numtasks - 1)) next = 0;
    
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
    
    MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
    
    { do some work }
    MPI_Waitall(4, reqs, stats);
    
    MPI_Finalize();
}
```
Example: Odd-Even Sort

```
3  2  3  8  5  6  4  1
2  3  3  8  5  6  1  4
2  3  3  5  8  1  6  4
2  3  3  5  1  8  4  6
2  3  3  1  5  4  8  6
2  3  1  3  4  5  6  8
2  1  3  3  4  5  6  8
1  2  3  3  4  5  6  8
1  2  3  3  4  5  6  8
```

Phase 1 (odd)

1. procedure ODD-EVEN_PAR(n)
2. begin
Phase 2 (even)

3. id := process’s label
4. for i := 1 to n do
5. begin
Phase 3 (odd)

6. if i is odd then
7. if id is odd then
8. compare-exchange_min(id + 1);
9. else
10. compare-exchange_max(id - 1);
Phase 4 (even)

11. if i is even then
12. if id is even then
13. compare-exchange_min(id + 1);
14. else
15. compare-exchange_max(id - 1);
Phase 5 (odd)

16. end for
Phase 6 (even)

17. end ODD-EVEN_PAR
Phase 7 (odd)

[From GGKK book Chapter 6.3.5]
Example: Odd-Even Sort

Compare-Split Operation
Example: Odd-Even Sort

```c
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */
main(int argc, char *argv[])
{
    int n; /* The total number of elements to be sorted */
    int npes; /* The total number of processes */
    int myrank; /* The rank of the calling process */
    int nlocal; /* The local number of elements */
    int *elmnts; /* The array that stores the local elements */
    int *relmnts; /* The array that stores the received elements */
    int oddrank; /* rank of the process during odd-phase communication */
    int evenrank; /* rank of the process during even-phase communication */
    int *wspace; /* Working space during the compare-split operation */
    int i;
    MPI_Status status;
```
/* Initialize MPI and get system information */
MPI_Init(&argc, &argv);
MPI_comm_size(MPI_COMM_WORLD, &npes);
MPI_comm_rank(MPI_COMM_WORLD, &myrank);

n = atoi(argv[1]);
nlocal = n/npes; /* Compute the number of elements to be stored locally. */

/* Allocate memory for the various arrays */
elmnts = (int *)malloc(nlocal*sizeof(int));
remlnts = (int *)malloc(nlocal*sizeof(int));
wspace = (int *)malloc(nlocal*sizeof(int));

/* Fill-in the elmnts array with random elements */
srandom(myrank);
for (i=0; i<nlocal; i++)
    elmnts[i] = random();

/* Sort the local elements using the built-in quicksort routine */
qsort(elmnts, nlocal, sizeof(int), IncOrder);
Example: Odd-Even Sort (cont’d)

/* Determine the rank of the processors that myrank needs to communicate during the odd and even phases of the algorithm */

if (myrank%2 == 0) {
    oddrank = myrank-1;
    evenrank = myrank+1;
}
else {
    oddrank = myrank+1;
    evenrank = myrank-1;
}

/* Set the ranks of the processors at the end of the linear */

if (oddrank == -1 || oddrank == npes)
    oddrank = MPI_PROC_NULL;

if (evenrank == -1 || evenrank == npes)
    evenrank = MPI_PROC_NULL;

/* `dummy` process, a communication with process MPI_PROC_NULL has no effect. */
/* Get into the main loop of the odd-even sorting algorithm */
for (i=0; i<npes-1; i++) {
    if (i%2 == 1) /* Odd phase */
        MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
                     nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
    else /* Even phase */
        MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
                     nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);
    CompareSplit(nlocal, elmnts, relmnts, wspace,
                 myrank < status.MPI_SOURCE);
}
free(elmnts);
free(relmnts);
free(wspace);
MPI_Finalize();
**Order and Fairness of MPI**

- **Order**: MPI guarantees that messages will not overtake each other.
  - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
  - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.

- **Fairness**: MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.