The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space.
- MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process’s address space to another’s
What is MPI?

• A message-passing library specification
  – extended message-passing model
  – not a language or compiler specification
  – not a specific implementation or product
• For parallel computers, clusters, and heterogeneous networks
• Full-featured
• Designed to provide access to advanced parallel hardware for
  – end users
  – library writers
  – tool developers
MPI Sources

• The Standard itself:
  – at http://www.mpi-forum.org
  – All MPI official releases, in both postscript and HTML

• Books:
  – Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995
  – Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997
  – MPI: The Complete Reference Vol 1 and 2, MIT Press, 1998(Fall)

• Other information on Web:
  – at http://www.mcs.anl.gov/mpi
  – pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages
Why Use MPI?

• MPI provides a powerful, efficient, and portable way to express parallel programs
• MPI was explicitly designed to enable libraries…
• … which may eliminate the need for many users to learn (much of) MPI
MPI: Brief History

- Standardizing process began in April 1992 in Virginia
- Project involved approximately 60 people from 40 organizations in U.S. and Europe
- Efforts were made to establish the basic features needed for a standard MPI
- Proposal put forward in November 1992
- Committee members met every 6 weeks for 2 days during a 9 month period.
- Final draft of MPI standard done by Fall 93
- MPI version 1.1 released in June 1995
- MPI version 2.0 released in July 1997
  - Basic additions include process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, and additional language bindings
A Minimal 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;
}
Running MPI

- In C:
  - mpi.h must be #included
  - MPI functions return error codes or **MPI_SUCCESS**

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.

- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
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?

• MPI provides functions to answer these questions:
  – `MPI_Comm_size` reports the number of processes.
  – `MPI_Comm_rank` reports the rank, a number between 0 and size-1, identifying the calling process
Better Hello (C)

#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;
}
Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*
- Data is explicitly *sent* by one process and *received* by another
- An advantage is that any change in the receiving process’s memory is made with the receiver’s explicit participation
- Communication and synchronization are combined

![Diagram of message passing between Process 0 and Process 1]

Process 0

| Send(data) |

Process 1

| Receive(data) |
One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2

```
Process 0          Process 1
Put(data)          (memory)
(memory)           Get(data)
```
MPI Basic Send/Receive

• We need to fill in the details in

  Process 0  Process 1
  
  Send(data)  Receive(data)

• Things that need specifying:
  – How will “data” be described?
  – How will processes be identified?
  – How will the receiver recognize/screen messages?
  – What will it mean for these operations to complete?
What is message passing?

- Data transfer plus synchronization

- Requires cooperation of sender and receiver
- Cooperation not always apparent in code
Some Basic Concepts

- 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*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called *MPI_COMM_WORLD*. 
MPI Datatypes

• The data in a message to be sent or received is described by a tuple (address, count, datatype)

• An MPI \textit{datatype} is recursively defined as:
  – predefined, corresponding to a data type from the language (e.g., MPI\_INT, MPI\_DOUBLE\_PRECISION)
  – a contiguous array of MPI datatypes
  – a strided block of datatypes
  – an indexed array of blocks of datatypes
  – an arbitrary structure of datatypes

• There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.
MPI Tags

• Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.

• Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive.

• Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes.
MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by \((\text{start, count, datatype})\).
- The target process is specified by \(\text{dest}\), which is the rank of the target process in the communicator specified by \(\text{comm}\).
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.
MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (on source and tag) message is received from the system, and the buffer can be used.
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE.
- status contains further information
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.
Retrieving Further Information

- **Status** is a data structure allocated in the user’s program.
- In C:

  ```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 );
  ```
Tags and Contexts

• Separation of messages used to be accomplished by use of tags, but
  – this requires libraries to be aware of tags used by other libraries
  – this can be defeated by use of “wild card” tags

• Contexts are different from tags
  – no wild cards allowed
  – allocated dynamically by the system when a library sets up a communicator for its own use

• User-defined tags still provided in MPI for user convenience in organizing application

• Use MPI_Comm_split to create new communicators
MPI is Simple

• Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  – MPI_INIT
  – MPI_FINALIZE
  – MPI_COMM_SIZE
  – MPI_COMM_RANK
  – MPI_SEND
  – MPI_RECV

• Point-to-point (send/recv) isn’t the only way...
Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- `MPI_BCAST` distributes data from one process (the root) to all others in a communicator.
- `MPI_REDUCE` combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, `SEND/RECEIVE` can be replaced by `BCAST/REDUCE`, improving both simplicity and efficiency.
Sources of Deadlocks

• Send a large message from process 0 to process 1
  – If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

• What happens with

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

- This is called “unsafe” because it depends on the availability of system buffers
Some Solutions to the “unsafe” Problem

• Order the operations more carefully:

  Process 0   Process 1
  \hline
  Send(1)     Recv(0)
  Recv(1)     Send(0)

• Use non-blocking operations:

  Process 0   Process 1
  \hline
  Isend(1)    Isend(0)
  Irecv(1)    Irecv(0)
  Waitall     Waitall
When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for others
  - Libraries
- Need to manage memory on a per processor basis
When *not* to use MPI

- Regular computation matches HPF
- Solution (e.g., library) already exists
- Require Fault Tolerance
  - Sockets
- Distributed Computing
  - CORBA, DCOM, etc.
Evaluating Parallel Programs
Execution Time

Sequential execution time, $t_s$: Estimate by counting computational steps of best sequential algorithm.

Parallel execution time, $t_p$: In addition to number of computational steps, $t_{\text{comp}}$, need to estimate communication overhead, $t_{\text{comm}}$:

$$t_p = t_{\text{comp}} + t_{\text{comm}}$$
Computational Time

Count number of computational steps. When more than one process executed simultaneously, count computational steps of most complex process. Generally, function of n and p, i.e.

\[ t_{\text{comp}} = f(n, p) \]

Often break down computation time into parts. Then

\[ t_{\text{comp}} = t_{\text{comp1}} + t_{\text{comp2}} + t_{\text{comp3}} + \ldots \]

Analysis usually done assuming that all processors are same and operating at same speed.
Communication Time

Many factors, including network structure and network contention. As a first approximation, use

\[ t_{\text{comm}} = t_{\text{startup}} + nt_{\text{data}} \]

\( t_{\text{startup}} \) is startup time, essentially time to send a message with no data. Assumed to be constant.

\( t_{\text{data}} \) is transmission time to send one data word, also assumed constant, and there are \( n \) data words.
Idealized Communication Time

![Graph showing Idealized Communication Time]

- **Time**
- **Number of data items (n)**
- **Startup time**
Final communication time, $t_{\text{comm}}$, the summation of communication times of all sequential messages from a process, i.e.

$$t_{\text{comm}} = t_{\text{comm}1} + t_{\text{comm}2} + t_{\text{comm}3} + \ldots$$

Typically, communication patterns of all processes same and assumed to take place together so that only one process need be considered.

Both startup and data transmission times, $t_{\text{startup}}$ and $t_{\text{data}}$, measured in units of one computational step, so that can add $t_{\text{comp}}$ and $t_{\text{comm}}$ together to obtain parallel execution time, $t_p$. 
Benchmark Factors

With $t_s$, $t_{\text{comp}}$, and $t_{\text{comm}}$, can establish speedup factor and computation/communication ratio for a particular algorithm/implementation:

\[
\text{Speedup factor} = \frac{t_s}{t_p} = \frac{t_s}{t_{\text{comp}} + t_{\text{comm}}}
\]

\[
\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}}
\]

Both functions of number of processors, $p$, and number of data elements, $n$.

Will give indication of scalability of parallel solution with increasing number of processors and problem size. Computation/communication ratio will highlight effect of communication with increasing problem size and system size.
Debugging and Evaluating Parallel Programs Empirically

Visualization Tools

Programs can be watched as they are executed in a space-time diagram (or process-time diagram):

![Space-time diagram](image-url)

Legend:
- **Purple**: Computing
- **Light purple**: Waiting
- **Red**: Message-passing system routine
- **Red arrow**: Message

Time
Implementations of visualization tools are available for MPI.

An example is the Upshot program visualization system.
Evaluating Programs Empirically
Measuring Execution Time

• To measure the execution time between point L1 and point L2 in the code, we might have a construction such as

L1: time(&t1); /*start timer*/
L2: time(&t2); /*stop timer*/
elapsed_time = difftime(t2,t1);
/*elapsed_time = t2 - t1*/
printf("Elapsed time = %5.2f seconds", elapsed_time);

• MPI provides the routine MPI_Wtime() for returning time (in seconds).

L1: tmptime = MPI_Wtime();
< do some computation >
L2: comp_time = MPI_Wtime() - tmptime;
if (me == 0)
printf("Comp time for proc. 0 = %5.2f seconds", comp_time);
Summary

• The parallel computing community has cooperated on the development of a standard for message-passing libraries
• There are many implementations, on nearly all platforms
• MPI subsets are easy to learn and use
• Lots of MPI material is available

Acknowledgments: W. Gropp, E. Lusk, Argonne National Laboratory