Writing Parallel Programs

- Directives-based data-parallel languages
  - Serial code is made parallel by adding directives (which appear as comments in the serial code) that tell the compiler how to distribute the data and work among the processors
  - Normally available for shared memory architectures because the global address space greatly simplifies the writing of compilers
  - Examples: OpenMP, HPF
- Explicit message passing
  - Programmer explicitly divides data and code and specifies communication among processors
  - Normally suitable for distributed memory architectures
  - Examples: MPI, PVM

Message Passing Model

- Consists of a number of processes, each working on some local data
- Each process has local variables, and it cannot directly access the variables of another process
- Sharing of data is done by explicitly sending and receiving messages

```
Process 0
   ...    Send(P1, msg1)
   ...    Recev(P0, msg1)
   ...
Process 1
   ...    Send(P0, msg2)
   ...    Recev(P1, msg2)
   ...
```

Characteristics of Message Passing Model

- General and flexible
  - Essentially any parallel computation can be cast in the message passing form
  - Can be implemented on a wide variety of architectures, including shared-memory, distributed-memory, and even single processor machines
  - Generally allows more control over data location and flow in a parallel computation
  - Both domain and functional decompositions can be cast in message passing form
- Tedious
  - Generally more tedious to program in message passing form

Message Passing Interface (MPI)

- MPI is a message-passing library
  - Provides an interface for communicating (exchange of data and synchronization of tasks) among the processors in a distributed memory architecture
- Standard and portable library: both the syntax and behavior of MPI is standardized.
- High performance: higher performing that earlier message-passing libraries
- Functionality: rich set of functions and behaviors are provided
- Multiplicity of implementations: Numerous implementations are available, including open-source implementations like LAM/MPI and MPICH

MPI Functions

- General
  - Functions for initiating and terminating communication
  - Functions for creating and managing groups of processors for communication
- Point-to-point communication
  - Functions for sending and receiving messages between two named processors
- Collective communication
  - Functions for exchanging data among a group of processors
  - Functions for synchronization
  - Functions for mathematical operations on data held by group of processors
**MPI Function Format**

For C:
- Header file: `#include <mpi.h>`
- `rc = MPI_Xxxxx(parameter, …)`
  - `rc` is return code of type integer
  - Case is important. MPI and first letter after underscore is capitalized
- Exceptions to this format are the timing functions (MPI_Wtime() and MPI_Wtick()) which return floating-point of type `double`

**Initializing and Terminating**

- Initializing MPI: must be first MPI function call before others
  - `int MPI_Init(int *argc, char ***argv)`
- Terminating MPI: must be the last MPI function call
  - `int MPI_Finalize()`

**Communicators**

- Programmer’s view: group of processes that are allowed to communicate with each other
- All MPI communication calls have a communicator argument
- Most often use MPI_COMM_WORLD
  - Defined by default by MPI_Init()
  - It includes all processors

**Rank and Size**

- Process ID number within the communicator, starting with zero
- Used to specify source and destination of messages
  - `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - How many processes are contained within a communicator?
  - `int MPI_Comm_size(MPI_Comm comm, int *size)`

**Example**

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

int main(int argc, char *argv[]) {
    int myid, p, err;
    err = MPI_Init(argc, argv);
    err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    err = MPI_Comm_size(MPI_COMM_WORLD, &p);
    printf("Hello! I am processor %d of %d\n", myid, p);
}
```

**Messages**

- Message consist of two parts, each made up of 3 parameters
  - Data: data that is being sent
  - Envelope: information that helps to route the data
  - Message = data (3 parameters) + envelope (3 parameters)
- Data
  - `data`: address where data start
  - `count`: number of elements from startbuf
  - `datatype`: type of data to be transmitted
- Envelope
  - `destination/source`: the rank or ID of receiving/sending process
  - `tag`: arbitrary number to distinguish among messages
  - `Communicator`: communicator in which communication is to take place
Basic Datatypes

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_BYTE</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>byte</td>
</tr>
</tbody>
</table>

Derived Datatypes

- User defined datatypes
  - Built up from basic datatypes
  - Depending on the nature of data to be communicated, using derived datatypes makes programming easier and can improve performance
  - Examples: contiguous, vector, struct
- Rules and rationale for MPI datatypes
  - Programmer declares variables to have “normal” C type, but uses matching MPI datatype as arguments in MPI functions
  - Mechanism to handle type conversion in a heterogeneous collection of machines
  - General rule: MPI datatype specified in a receive must match the MPI datatype specified in the send

Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator
- Destination process is identified by its rank in the communicator

Definitions

- “Completion” means that memory locations used in the message transfer can be safely accessed
  - send: variable sent can be reused after completion
  - receive: variable received can now be used
- MPI communication modes differ in what conditions on the receiving end are needed for completion
- Communication modes can be blocking or non-blocking
  - Blocking: return from function call implies completion
  - Non-blocking: routine returns immediately, completion tested for

Communication Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Completion Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has completed</td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless an error occurs), irrespective of receiver</td>
</tr>
<tr>
<td>Standard send</td>
<td>Message sent (receive state unknown)</td>
</tr>
<tr>
<td>Ready send</td>
<td>Always completes (unless an error occurs), irrespective of whether the receive has completed</td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a message has arrived</td>
</tr>
</tbody>
</table>