MPI and – Message passing interface
(Chapter 3)

Introduction to MPI
(https://computing.llnl.gov/tutorials/mpi/)

• All large scale multiprocessors have “physically” distributed memory systems.
• A lot of overhead when building a shared address space on top of a physically distributed memory system.
• Some problems can naturally be partitioned into parallel sub-problems (with possible coordination and synchronization)
• MPI (Message Passing Interface) evolved as the standard interface for message passing libraries.
• Note: Sockets is Unix’ way of passing messages and many MPI libraries are built using sockets. MPI, however, is much easier to use than sockets.
• An MPI implementation allows a user to start multiple threads (SPMD programming style) and provide functions for the threads to communicate and synchronize.
SPMD Programs

- The user specifies the number of processes and number of processors.
- The same source code is executed by all processes
- One or more process can execute on each processor
- The set of processes is defined as the “MPI_COMM_WORLD”
- Can have different processes do different things by using the process id (rank)
  - MPI_Comm_rank(MPI_COMM_WORLD, &rank)
- Subsets of MPI_COMM_WORLD, called communicators, can be defined by the user.
  - Rami’s_world
    - MPI_Comm_rank(Rami’s_world, &rank)

A simple MPI Program

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
    int numtasks, my_rank, rc;
    rc = MPI_Init(&argc,&argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program \n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    if (my_rank == 0) { /* master */
        printf("#of tasks= %d, My rank= %d\n",numtasks,rank);
    } else { /* worker */
        printf("My rank= %d\n", rank);
    }
    MPI_Finalize();
}
```

Has to be called first, and once

Has to be called last, and once
Point-to-Point Communication

- Sending process
- Kernel
- Network
- Kernel
- Receiving process

Path of a message across address spaces

Can be explicitly allocated (in buffered send/receive)

Blocking Point-to-Point Communication

Address of data (usually variable name)

MPI_Send(x, #_of_items, item_type, dest_rank, tag, communicator);

MPI_Recv(x, #_of_items, item_type, source_rank, tag, communicator, &status);

Predefined: MPI_CHAR, MPI_INT, MPI_FLOAT,...

A structure of type MPI_Status

MPI_Send Source_rank

MPI_Recv Dest_rank

Blocking: Return after the sender application buffer is free for reuse, or the application buffer received the message, respectively.
Out of order receiving

\texttt{MPI\_Recv(x, MAX\_items, item\_type, MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, communicator, \&status);}  
Larger or equal to expected size  
\texttt{MPI\_Status* \{MPI\_SOURCE, MPI\_TAG, MPI\_ERROR\}}  
Allows message reception from any source  
\texttt{MPI\_Get\_count(MPI\_Status status /*in*/, MPI\_Datatype type /*in*/, int* count /*out*/);}  
Get actual values using

Non-blocking Point-to-Point Communication

\texttt{MPI\_Isend(x, \#\_of\_items, item\_type, dest\_rank, tag ,communicator, \&request);}  
A request number returned by MPI. Of type MPI\_Request  
\texttt{MPI\_Irecv(x, \#\_of\_items, item\_type, source\_rank, tag, communicator, \&request);}  
\texttt{MPI\_Wait(&request, \&status);}  
Blocks until the operation corresponding to “request” is completed  
\texttt{MPI\_Waitall(count, array of requests, array of statuses);}  
\texttt{MPI\_Test(&request, \&flag, \&status);}  
\texttt{MPI\_Testall();}  
\texttt{MPI\_Testsome();}  
\texttt{MPI\_Testany();}  
non-blocking  
Returns “true” (1) if operation had completed and “false” (0), otherwise
Types of send/receive

- **Blocking**: MPI_Send() and MPI_Recv()
  - Return after the sender application buffer is free for reuse, or the application buffer received the message, respectively.

- **Synchronous blocking**: MPI_Ssend()
  - Returns after the destination process received the message

- **Non-blocking**: MPI_Isend() and MPI_Irecv()
  - Returns immediately. MPI_wait and MPI_Test indicate that the non-blocking send or receive has completed locally

- **Synchronous non-blocking**: MPI_Issend()
  - Returns immediately. MPI_wait and MPI_Test indicate that the destination process has received the message

- **Buffered**: allows the programmer to explicitly control system buffers.

Example – The trapezoidal rule for integration

```c
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
x_i = a + i*h;
approx += f(x_i);
}
approx = h*approx;
```
Dealing with input

Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin. Hence, it must read the data and send to the other processes.

Bad practice to depend on in-order message delivery. Should use tags
Type of messages

- **Point-to-point**: one processor sends a message to another processor
- **One-to-all**: one processor broadcasts a message to all other processors
- **One-to-all personalized**: one processor sends a different message to each other processor
- **All-to-all**: each processor broadcasts a message to all other processors
- **All-to-all personalized**: each processor sends a different message to each other processors

Collective communication

- Can be built using point-to-point communications, but typical MPI implementations have optimized them
- All processes place the same call, although depending on the process, some arguments may not be used

- **MPI_Bcast**\((x, n\text{-}items, \text{type}, \text{root}, \text{MPI\_COMM\_WORLD})\)
- **MPI_Barrier**\((\text{MPI\_COMM\_WORLD})\)
- **MPI_Reduce**\((x, r, n\text{-}items, \text{type}, \text{op}, \text{root}, \text{MPI\_COMM\_WORLD})\)
  - Private data to be reduced
  - Location of reduced data
  - Operator used in reduction: MPI\_MAX, MPI\_SUM, MPI\_PROD, ...
- **MPI_Allreduce**\((x, r, n\text{-}items, \text{type}, \text{op}, \text{MPI\_COMM\_WORLD})\)
  - Same as MPI\_Reduce() except that every thread gets the result, not only “root” (equivalent to MPI\_Reduce followed by MPI\_Bcast)
Efficiency of MPI_Allreduce

A global sum followed by distribution of the result.

A butterfly-structured (hypercube) global sum.

Order of collective Communication

- Collective communications do not use tags – they are matched purely on the basis of the order in which they are called
- The names of the memory locations are irrelevant to the matching

- Example: Assume three processes with calling MPI_Reduce with operator MPI_SUM, and destination process 0.

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a = 1; c = 2</td>
<td>a = 1; c = 2</td>
<td>a = 1; c = 2</td>
</tr>
<tr>
<td>1</td>
<td>MPI_Reduce(a, b, ...)</td>
<td>MPI_Reduce(c, d, ...)</td>
<td>MPI_Reduce(a, b, ...)</td>
</tr>
<tr>
<td>2</td>
<td>MPI_Reduce(c, d, ...)</td>
<td>MPI_Reduce(a, b, ...)</td>
<td>MPI_Reduce(c, d, ...)</td>
</tr>
</tbody>
</table>

- The order of the calls will determine the matching so, in process 0, the value stored in b will be 1+2+1 = 4, and the value stored in d will be 2+1+2 = 5.
**Scatter (personalized broadcast – one to many)**

\[
\text{MPI\_Scatter}(s, n, s, s\text{-type}, r, n, r, r\text{-type}, \text{root}, \text{MPI\_COMM\_WORLD})
\]

Data to be scattered, \# of Items sent, Location of scattered data, \# of Items received by each thread

![Diagram of Scatter operation]

---

**Scatter Example**

```c
int main(int argc, char **argv) {
    int *a;
    double *recvbuffer;
    ...
    MPI_Comm_size(MPI_COMM_WORLD,&n);
    if (my_rank == 0) { /* master */
        <allocate array a of size N>
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(a, N/n, MPI_INT, recvbuffer, N/n, MPI_INT, 0, MPI_COMM_WORLD);
    } else { /* worker */
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(NULL, 0, MPI_INT, recvbuffer, N/n, MPI_INT, 0, MPI_COMM_WORLD);
    }
    ...
}
```

Can use \text{MPI\_IN\_PLACE}

---

17

18
Gather (many to one)

\text{MPI\_Gather}(s, n_s, s\_type, r, n_r, r\_type, \text{root}, \text{MPI\_COMM\_WORLD})

Data to be gathered

Location of gathered data

Proc. 0 (root)
Proc. 1
Proc. 2
Proc. 3

\text{MPI\_Allgather}(s, n_s, s\_type, r, n_r, r\_type, \text{MPI\_COMM\_WORLD})

Examples of global-local data mapping

- Consider \( n \times n \) matrix/vector multiplication \( y = A \times x \) on \( P \) processors.
- To minimize communication, partition \( A \) and \( y \) row wise.

- Each processor, \( \text{pid} \), will allocate two \( k = n/P \) vectors for its shares of \( x \) and \( y \) and an \( k \times n \) matrix (call it \( \text{local}_A[] \)) for its share of \( A \).

\[
\begin{align*}
\text{local}_A[i,j] &= A[k*\text{pid} + i, j] \\
\text{local}_y[i] &= A[k*\text{pid} + i]
\end{align*}
\]

- In SOR (Laplace iterative solver), we may simplify programming by augmenting the local domains by a stripe to accommodate boundary data received from other processors.
Example: Matrix-vector multiplication

```c
void Mat_vect_mult(
    double local_A[] /* in */,
    double local_x[] /* in */,
    double local_y[] /* out */,
    int local_n /* in */,
    int n /* in */,
    int local_n /* in */,
    MPI_Comm comm /* in */) {
    double *x;
    int local_i, j;
    int local ok = 1;
    x = malloc(n*sizeof(double));
    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
        x, local_n, MPI_DOUBLE, comm);

    for (local_i = 0; local_i < local_n; local_i++) {
        local_y[local_i] = 0.0;
        for (j = 0; j < n; j++)
            local_y[local_i] += local_A[local_i*n+j]*x[j];
    }
    free(x);
} /* Mat_vect_mult */
```

All to all personalized

```c
MPI_Alltoall(s, n_s, s_type, r, n_r, r_type, MPI_COMM_WORLD)
```

Example: Matrix transpose
Derived data types

- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- This allows the use of these data types in the send and receive calls.
- Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.
- Trapezoidal Rule example:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
</tr>
<tr>
<td>n</td>
<td>48</td>
</tr>
</tbody>
</table>

\{(MPI\_DOUBLE, 0), (MPI\_DOUBLE, 16), (MPI\_INT, 24)\}

What more can you do?

- Build virtual topologies
- Define new communicators from MPI\_COMM\_WORLD

- Extract handle of old group
  - MPI\_Comm\_group ()
- Form new group as a subset of old group
  - MPI\_Group\_incl ()
- Create new communicator for new group
  - MPI\_Comm\_create ()
- Determine new rank in new communicator
  - MPI\_Comm\_rank ()
- Communicate in new group
- Free up new communicator and group
  - MPI\_Comm\_free ()
  - MPI\_Group\_free ()
Overlapping communication and computation

- Example in SOR can
  - Isend
  - Ireceive
  - Do computation that do not depend on received message
  - Wait for receive to complete
  - Complete the computation.