Recitation 9

Ammar Karkour

October 6, 2022
Logistics

• P2 done (almost)
• P3 out (discussion next week)
Parallel Programming Models

- Shared Memory Model
- Message Passing Model
Parallel Programming Models

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
</table>

# Parallel Programming Models

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communicating processes usually reside on the same machine</td>
<td>Typically used in a distributed environment where communicating processes reside on remote machines connected through a network.</td>
</tr>
<tr>
<td>Faster communication strategy.</td>
<td>Relatively slower communication strategy</td>
</tr>
<tr>
<td>More difficult to synchronize</td>
<td>Easier to synchronize</td>
</tr>
<tr>
<td>Example: OpenMP</td>
<td>Example: MPI</td>
</tr>
</tbody>
</table>
What is MPI?

• Message Passing Interface

• Defines a set of API declarations on message passing (such as send, receive, broadcast, etc.), and what behavior should be expected from the implementations.

• The *de-facto* method of writing message-passing applications

• Applications can be written in C, C++ and calls to MPI can be added where required
MPI Program Skeleton

1. Include MPI Header File
2. Start of Program
   (Non-interacting Code)
3. Initialize MPI
4. Run Parallel Code & Pass Messages
5. End MPI Environment
   (Non-interacting Code)
6. End of Program

Photo credits:
MPI Program Skeleton

```c
#include <mpi.h>

int main (int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    // Run parallel code
    MPI_Finalize(); // End MPI Environment
    return 0;
}
```
MPI Concepts

• **Communicator**
  • Defines which collection of processes may communicate with each other to solve a certain problem
  • In this collection, each process is assigned a unique *rank*, and they explicitly communicate with one another by their ranks.
  • When an MPI application starts, it automatically creates a communicator comprising all processes and names it **MPI_COMM_WORLD**

• **Rank**
  • Within a communicator, every process has its own unique ID referred to as *rank*
  • Ranks are used by the programmer to specify the source and destination of messages
MPI Concepts

- Ranks within MPI_COMM_WORLD are printed in red
- Ranks within Comm_Fluid are printed in green
- Ranks within Comm_Struct are printed in blue
## MPI Concepts

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Init(int *argc, char ***argv)</code></td>
<td>Initialize the MPI library (must be the first routine called)</td>
</tr>
<tr>
<td><code>MPI_Comm_rank(comm, &amp;rank);</code></td>
<td>Returns the rank of the calling MPI process within the communicator, <code>comm</code></td>
</tr>
<tr>
<td><code>MPI_Comm_size(comm, &amp;size)</code></td>
<td>Returns the total number of processes within the communicator, <code>comm</code></td>
</tr>
<tr>
<td><code>MPI_COMM_WORLD</code></td>
<td>is set during <code>Init(...)</code></td>
</tr>
<tr>
<td>Other communicators can be created if needed</td>
<td></td>
</tr>
</tbody>
</table>
Let’s write our first MPI program...
MPI Send and Recv

MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

• The first argument is the data buffer
• The second and third arguments describe the count and type of elements that reside in the buffer
• MPI Datatype is very similar to a C datatype: MPI_INT, MPI_CHAR
• The fourth and fifth arguments specify the rank of the sending/receiving process and the tag of the message
• The sixth argument specifies the communicator

MPI_Recv (void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)

Why do we need a tag?
Let’s look at some parallel programs
Collective Communication

• Collective communication allows you to exchange data among a group of processes
• It must involve all processes in the scope of a communicator
• Hence, it is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operation
Patterns of Collective Communication

1. Broadcast
Patterns of Collective Communication

- Broadcasts a message from the process with rank root to all other processes of the group

```c
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```
Patterns of Collective Communication

MPI_Scatter(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)

• Distributes elements of sendbuf to all processes in comm
• Although the root process (sender) contains the entire data array, MPI_Scatter will copy the appropriate element into the recvbuf of the process.
• sendcount and recvcount are counts per process
Patterns of Collective Communication

**MPI_Gather**

```c
MPI_Gather(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- Inverse of MPI_Scatter
- Only the root process needs to have a valid receive buffer. All other calling processes can pass NULL for recv_data
Patterns of Collective Communication

- Reduces values on all processes within a group.
- The `sendbuf` parameter is an array of elements of type `datatype` that each process wants to reduce.
- The `recvbuf` is only relevant on the process with a rank of `root`.
- The `recvbuf` array contains the reduced result and has a size of `sizeof(datatype) * count`.
- The `op` parameter is the operation that you wish to apply to your data.
- `MPI` contains a set of common reduction operations that can be used.

```c
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

Why not just `sizeof(datatype)`?
Patterns of Collective Communication
Patterns of Collective Communication

1. Broadcast
2. Scatter
3. Gather
4. Allgather
5. Alltoall
6. Reduce
7. Allreduce
8. Scan
9. Reducescatter
Let’s implement a more efficient parallel_sum