Message Passing and MPI
Abhinav Bhavele, Department of Computer Science
Announcements

• If you registered for the course recently, please email the TAs for a deepthought2 account

• Assignment 0 will be posted on Sept 20 and will be due on Sept 27
  • Not graded, 0 points
Shared memory architecture

- All processors/cores can access all memory as a single address space

Uniform Memory Access

Non-uniform Memory Access (NUMA)

https://computing.llnl.gov/tutorials/parallel_comp/#SharedMemory
Distributed memory architecture

- Each processor/core only has access to its local memory
- Writes in one processor’s memory have no effect on another processor’s memory

Non-uniform Memory Access (NUMA)  Distributed memory
Distributed memory architecture

- Each processor/core only has access to its local memory
- Writes in one processor’s memory have no effect on another processor’s memory

Non-uniform Memory Access (NUMA)  Distributed memory
Programming models

- **Shared memory model:** All threads have access to all of the memory
  - Pthreads, OpenMP

- **Distributed memory model:** Each process has access to their own local memory
  - Also sometimes referred to as message passing
  - MPI, Charm++

- **Hybrid models:** Use both shared and distributed memory models together
  - MPI+OpenMP, Charm++ (SMP mode)
Distributed memory programming models

- Each process only has access to its own local memory / address space
- When it needs data from remote processes, it has to send messages
Message passing

- Each process runs in its own address space
  - Access to only their memory (no shared data)
- Use special routines to exchange data
Message passing programs

• A parallel message passing program consists of independent processes
  • Processes created by a launch/run script

• Each process runs the same executable, but potentially different parts of the program

• Often used for SPMD style of programming
Message passing history

- PVM (Parallel Virtual Machine) was developed in 1989-1993

- MPI forum was formed in 1992 to standardize message passing models and MPI 1.0 was released around 1994
  - v2.0 - 1997
  - v3.0 - 2012
Message Passing Interface (MPI)

- It is an interface standard — defines the operations / routines needed for message passing
- Implemented by vendors and academics for different platforms
  - Meant to be “portable”: ability to run the same code on different platforms without modifications
- Some popular implementations are MPICH, MVAPICH, OpenMPI
Hello world in MPI

#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("Hello world! I'm %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
Compiling and running an MPI program

• Compiling:

  mpicc -o hello hello.c

• Running:

  mpirun -n 2 ./hello
Process creation / destruction

- int MPI_Init( int argc, char **argv )
  - Initializes the MPI execution environment
- int MPI_Finalize( void )
  - Terminates MPI execution environment
Process identification

- `int MPI_Comm_size( MPI_Comm comm, int *size )`
  - Determines the size of the group associated with a communicator

- `int MPI_Comm_rank( MPI_Comm comm, int *rank )`
  - Determines the rank (ID) of the calling process in the communicator

- **Communicator — a set of processes**
  - Default communicator: `MPI_COMM_WORLD`
Send a message

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

**buf**: address of send buffer

**count**: number of elements in send buffer

**datatype**: datatype of each send buffer element

**dest**: rank of destination process

**tag**: message tag

**comm**: communicator
Receive a message

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

- `buf`: address of receive buffer
- `status`: status object
- `count`: maximum number of elements in receive buffer
- `datatype`: datatype of each receive buffer element
- `source`: rank of source process
- `tag`: message tag
- `comm`: communicator
Simple send/receive in MPI

```c
int main(int argc, char *argv) {
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int data;
    if (rank == 0) {
        data = 7;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data %d from process 0\n", data);
    }

    ...
}
```
Basic MPI_Send and MPI_Recv

- MPI_Send and MPI_Recv routines are blocking
  - Only return when the buffer specified in the call can be used
  - Send: Returns once sender can reuse the buffer
  - Recv: Returns once data from Recv is available in the buffer
Basic MPI_Send and MPI_Recv

- MPI_Send and MPI_Recv routines are blocking
  - Only return when the buffer specified in the call can be used
  - Send: Returns once sender can reuse the buffer
  - Recv: Returns once data from Recv is available in the buffer

**Diagram:**

- Process 0
  - MPI_Send
  - Deadlock!
  - MPI_Recv

- Process 1
  - Time
Basic MPI_Send and MPI_Recv

- MPI_Send and MPI_Recv routines are blocking
  - Only return when the buffer specified in the call can be used
  - Send: Returns once sender can reuse the buffer
  - Recv: Returns once data from Recv is available in the buffer

```
Process 0 [ ] [ ] [ ]
Process 1 [ ] [ ] [ ]
Time

Deadlock!
```

```
MPI_Send
MPI_Recv
```
Example program

```c
int main(int argc, char *argv) {
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ...
    if (rank % 2 == 0) {
        data = rank;
        MPI_Send(&data, 1, MPI_INT, rank+1, 0, ...);
    } else {
        data = rank * 2;
        MPI_Recv(&data, 1, MPI_INT, rank-1, 0, ...);
    }
    ...
    printf("Process %d received data %d\n", data);
}
...
Example program

```c
int main(int argc, char *argv) {
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ...
    if (rank % 2 == 0) {
        data = rank;
        MPI_Send(&data, 1, MPI_INT, rank+1, 0, ...);
    } else {
        data = rank * 2;
        MPI_Recv(&data, 1, MPI_INT, rank-1, 0, ...);
    }
    ...
    printf("Process %d received data %d\n", data);
}
...
Example program

```c
int main(int argc, char *argv) {
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ...
    if (rank % 2 == 0) {
        data = rank;
        MPI_Send(&data, 1, MPI_INT, rank+1, 0, ...);
    } else {
        data = rank * 2;
        MPI_Recv(&data, 1, MPI_INT, rank-1, 0, ...);
    }
    ...
    printf("Process %d received data %d\n", data);
    }
    ...
}
# Example program

```c
int main(int argc, char *argv) {
    
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    
    if (rank % 2 == 0) {
        data = rank;
        MPI_Send(&data, 1, MPI_INT, rank+1, 0, ...);
    } else {
        data = rank * 2;
        MPI_Recv(&data, 1, MPI_INT, rank-1, 0, ...);
    }

    printf("Process %d received data %d\n", data);
}
```
Example program

```c
int main(int argc, char *argv) {
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ...
    if (rank % 2 == 0) {
        data = rank;
        MPI_Send(&data, 1, MPI_INT, rank+1, 0, ...);
    } else {
        data = rank * 2;
        MPI_Recv(&data, 1, MPI_INT, rank-1, 0, ...);
        ...
        printf("Process %d received data %d\n", data);
    }
    ...
}
```
MPI communicators

- Communicator represents a group or set of processes numbered 0, … , n-1
- Every program starts with MPI_COMM_WORLD (default communicator)
  - Defined by the MPI runtime, this group includes all processes
- Several MPI routines to create sub-communicators
  - MPI_Comm_split
  - MPI_Cart_create
  - MPI_Group_incl
MPI datatypes

- Can be a pre-defined one: MPI_INT, MPI_CHAR, MPI_DOUBLE, ...
- Derived or user-defined datatypes:
  - Array of elements of another datatype
  - struct data type to accommodate sending multiple datatypes
Abhinav B hattele
5218 Brendan Iribe Center (IRB) / College Park, MD 20742
phone: 301.405.4507 / e-mail: bhatele@cs.umd.edu