Announcement

Schedule for remainder of term

- Monday, 7 March: lecture and course evaluations
- Tuesday, 8 March: final exam prep
- Wednesday, 9 March: no lecture – work on final project
- Friday, 10 March: no lab – work on final project
- Thursday, 17 March: final exam, 8-10am
  - If you score higher on the final exam than you did on the midterm exam, the final exam score will replace your midterm exam score.

- Extra credit
  - Submit a single question (and answer)
  - I’ll make all of the submissions available to help you prepare for the final exam
  - One of the submitted questions will appear on the actual final exam
From last time

**Heavyweight processes**

**Lightweight processes**

**Q:** What is the difference between a heavyweight and lightweight process?
From last time

Heavyweight processes

Lightweight processes

Thread 1

Thread 2

Thread 3

get()
Q: Is the above an example of synchronous or asynchronous message passing?
From last time

Synchronous versus Asynchronous message passing

P1

Send()
Suspend()
Transfer()

request
ack
message

P2

Receive()
Suspend()

P1

gather()

P2

P3

...
Today

Message Passing

MPI
Scenario: There are 5 processes, each of which has a locally scoped variable. The goal is for EVERY process to learn of the smallest and largest of the locally scoped variables among the 5 processes.
Scenario: There are 5 processes, each of which has a locally scoped variable. The goal is for EVERY process to learn of the smallest and largest of the locally scoped variables among the 5 processes.

Q: What is the fewest number of messages that must be sent among the 5 processes (and in which order), to achieve the goal?

(on the board explanation)
Q: Which approach is best?

Centralized

Symmetric
Which of these two allows for the **fastest** run time (requires least amount of time from start to finish)?

Assume dedicated paths, and all processes are capable of running concurrently.

**Topologies**

- **Centralized**

  - Time to send: 2s
  - Time to sort: 10s

- **Symmetric**

  - Time to send: 10s
  - Time to sort: 2s
Topologies

How much work (sum of all work done by individual processes) is required?

Assume dedicated paths, and all processes are capable of running concurrently

Centralized

Symmetric

Time to send : 2s    Time to sort : 10s
Time to send : 10s   Time to sort : 2s
Can we assume that messages are sent and received in parallel?
Can we assume that messages are sent and received in parallel?

As $n$ becomes large, dedicated paths cannot be assumed. How does that affect your choice of topology in terms of # of messages that are sent, and time needed?
Can we assume that messages are sent and received in parallel?

As $n$ becomes large, dedicated paths cannot be assumed. How does that affect your choice of topology in terms of # of messages that are sent, and time needed?

(on the board discussion)
MPI = Message passing Interface

Lab 8
MPI = Message passing Interface

**MPI_Init**: initialize the MPI library

**MPI_COMM_WORLD**: The “world”

**MPI_Comm_size**: the number of processes started

**MPI_Comm_rank**: Determine your rank (unique ID)

**MPI_Send**: send another message. Includes the message, as well as recipient

**MPI_Receive**: receive a message from another process. Specifies into which buffer the data should be received

**MPI_Finalize**: terminate (knowledge of IDs is needed to terminate all processes)
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Parallel Region

    MPI_Finalize();
}
```
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    // Parallel Region

    MPI_Finalize();
}
```

Every instance of the program (every Processors is running this code) gets a copy of the arguments with which the program was started
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

// Parallel Region

MPI_Finalize();
}
```

Determine the number of processes working on “this” job, and what is “my” ID
MPI

MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Parallel Region

    MPI_Finalize();
}
```

After the parallel regions are “done,” then everything is cleaned up.

Q: How does MPI enable all processes to coordinate with each other?
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Parallel Region
    MPI_Finalize();
}
```

After the parallel regions are “done,” then everything is cleaned up.

There are message that enable: barriers, bcast, scatter, gather, reduce, etc.
Recall that MPI is a heavyweight process system, so ALL processes have the entirety of this program. Thus if you want a process to be a coordinator ...
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Parallel Region

    MPI_Finalize();
}
```

The parallel regions usually has code for a coordinator and a worker, and the process ID (rank) is used to distinguish the coordinator.
MPI = Message passing Interface

```c
if (my_rank == 0) {
    // receive
    // calculate
}
else {
    // calculate
    // send
}
```
Exam Prep