CS 220: Introduction to Parallel Computing

MPI: Sending/Receiving Messages

Lecture 14

Off Topic: The IOCCC

- The International Obfuscated C Code Contest is a celebration of the craziness of C
- Using the preprocessor and C hacks, contestants submit programs that look like one thing and do another
  - Or maybe just look like something… ASCII art style
- http://www.ioccc.org/

Today's Agenda

- ssh setup
- MPI Review
- MPI_Send and MPI_Recv
- I/O Buffering and Blocking

Today's Agenda

- **ssh setup**
- MPI Review
- MPI_Send and MPI_Recv
- I/O Buffering and Blocking

Setting up SSH

- A short guide is available on the schedule page
- You should be able to type ‘ssh <machine>’ and be logged in without a password
- Things to know:
  - ssh-keygen utility
  - Your ~/.ssh/authorized-keys file
- Let’s do this now

Today's Agenda

- ssh setup
- **MPI Review**
- MPI_Send and MPI_Recv
- I/O Buffering and Blocking

Functions we Learned Last Class

```
/* Total number of processes in this MPI communicator */
int comm_sz;
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

/* Get the rank of this processor */
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* Get the host name of this processor */
char hostname[MPI_MAX_PROCESSOR_NAME];
int name_sz;
MPI_Get_processor_name(hostname, &name_sz);
```

mpicc

- Instead of our usual **gcc** command, we use mpicc to compile MPI programs
- Recall the stages of compilation:
  1. Preprocessing
  2. Translation
  3. Linking
- For step 3, mpicc **links** against the MPI library
  - Dynamic linking

Moving On

- So far, all we’ve really done is ran several processes in parallel (all at the same time)
- The processes don’t talk, they just print their message and clean up
- We could do this on a single machine without MPI
  - We could also run our programs on multiple machines using ssh
- To **really** benefit from MPI, we need to actually pass messages!

Today's Agenda

- ssh setup
- MPI Review
- **MPI_Send and MPI_Recv**
- I/O Buffering and Blocking

MPI_Send

```c
char buffer[100];

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

- buf – address of the send buffer (first element)
- count – number of elements in send buffer
- datatype – kind of data in the buffer
- dest – rank of the destination
- tag – custom message tag
- comm – MPI communicator

MPI_Recv

```
char buffer[100];

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

- buf [OUT] – address of the **receive** buffer
- status [OUT] – information about the sender (rank, tag, length)
- The rest of the parameters are the same as MPI_Send

MPI Data Types (1/2)

- Note that we have to specify a data type to send/receive
- A few helpful types:
  - MPI_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_UNSIGNED_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - MPI_LONG_DOUBLE

MPI Data Types (2/2)

- Why would we need to specify these data types? Doesn't C already know what we're sending?
- Recall our MPI_Send/Recv functions, arg 1:
  - `void *buf`
- We're passing in a **void pointer**
  - a "generic" pointer to **any** data type

Revisiting C Arguments

- C supports variable length args
  - Remember our distinction between main() and main(void)?
- It does **not** support C++/Java/Python style **function overloading**
- So we have a few solutions:
  - printf() style where we embed the types in the format string or arguments. This is what MPI does.
  - Naming functions for each type – e.g., print_double()
  - Preprocessor macros – limited use

Compatibility

- MPI is supported by several programming languages, like C++, Fortran, etc.
- Limiting the scope of the data types helps ensure the library will be compatible with other languages
- Different architectures have different ways to organize data in memory
  - Big vs little endianness

Source

- When we receive data, we can specify the source rank
  - This lets us wait for process 1, then 2, etc… Or perhaps you're waiting to hear specifically from process 682.
- We can also use MPI_ANY_SOURCE to accept a message from any rank

```
char buffer[100];

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

Tags

- Since your programs may send and receive different types of messages, the 'tag' lets you identify them
- #define HAPPY_TAG 1
- #define SAD_TAG 2
- This lets you make sure you are receiving the message type you'd expect
  - MPI_Recv won't work if it receives a different tag
- You can also accept any tag with MPI_Recv by passing in MPI_ANY_TAG

Communicators

- Recall that MPI communicators are just a way to group processes
  - MPI_COMM_WORLD – all processes
- This functionality makes it easy to send messages selectively to particular processes
- For now, we'll just use MPI_COMM_WORLD

Status

- If you pass in an MPI_Status struct to MPI_Recv, it will be populated with information about the sender
- This can be useful, but we often don't need to worry about the sender
  - Generally we're more worried about actually processing the message
- If we don't care about the status info, we can pass in MPI_STATUS_IGNORE

Hello to the Next Level

- Given this, how can we enhance our hello world application to:
- Send the hello messages all to one process
- Communicate in a chain (pass the messages on)

Today's Agenda

- ssh setup
- MPI Review
- MPI_Send and MPI_Recv
- **I/O Buffering and Blocking**

Buffering

- When calling MPI_Send, MPI may decide to **buffer** the operation
- The message contents are copied into a buffer managed by MPI
  - Kind of like doing a `strcpy(dest, src)`
- The function returns immediately!
  - In other words, nothing has been sent but your program goes on to the next line
  - This is an **asynchronous** or **buffered** send

Synchronous Send

- We are used to synchronous functions in C
  1. Call the function
  2. It does its work
  3. **Then** it finally returns
- Upside: no buffering required here
  - Reduces memory consumption
- Downside: if the next steps in our program are printing "hello world" or computing pi, do we really need to wait for the message to reach its destination?

Standard Send

- The MPI_Send we've seen is a **standard send**
- It decides whether or not the operation should be buffered
  - MPI tries to choose the option that gives best performance
- To determine this, a **cut off** size is used
  - Message less than the cut off? Buffer it
  - Too big? Send it synchronously

Receiving Data

- MPI_Recv is considered a **blocking** call
- When you use MPI_Recv, it will wait until data arrives before doing anything
- This is kind of like our programs that use **scanf**
  - The function waits until we actually type a line before it resumes execution

Monitoring Blocked Processes

- We can see what processes are doing on our machine with the **top** command
- On Linux, we have a nice status column:
  - D uninterruptible sleep
  - R running
  - S sleeping (in the **blocked** state)
  - T stopped
  - Z zombie