Due by: **Tue, April 29 2016 BY 11:59 PM (EST)**

High Performance Computing

Homework #6: Part A

Due: Tuesday April 26 2016 by 11:59 PM

Email-based help Cutoff: 5:00 PM on Mon, April 25 2016

Maximum Points: 11

Submission Instructions

This homework assignment must be turned-in electronically via Canvas. Type in your responses to each question (right after the question in the space provided) in this document. You may use as much space as you need to respond to a given question. Once you have completed the assignment upload:

1. The MS-Word document (duly filled) and saved as a PDF file named with the convention `MUid.pdf` (example: **`raodm.pdf`**)

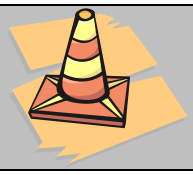
Note that copy-pasting from electronic resources, including lecture slides, is plagiarism. Consequently, you must suitably paraphrase the material in your own words when answering the following questions.

Name: Henry Ni

Objective

The objective of this homework is to review the necessary background information about distributed memory parallelism and MPI.

Read Section 6.1 and 6.2 from E-book "Introduction to Parallel Computing" (all students have free access to the electronic book). Links available off Syllabus page on Canvas.



Although the Safari E-books are available to all students there are only a limited number of concurrent licenses to access the books. Consequently, do not procrastinate working on this homework or you may not be able to access the E-books due to other users accessing books.

1. Describe two advantages and two disadvantages of the message passing model (not MPI) [**2 points**]
   a. 2 Advantages

   The message passing model for parallelism has its advantages over shared memory, specifically allowing the programmer greater control in determining memory use for parallel applications as well as being very portable (no special hardware is required).

   b. 2 Drawbacks

   Programmer has to be aware of memory use and ordering of communication. Load balancing must be done manually and can often result in poor performance if done improperly.

2. Discuss 2 significant differences between blocking buffered versus non-buffered operations [**1 points**]

Blocking buffered send/recv	***Blocking non-buffered send/recv***
No idle time since messages go to buffer before other process receives messages	Long blocking times between send/receive if two processes do not complete a handshake immediately
Potential risk in buffer overflow, lose data	Memory efficient (no buffers), no additional overhead

3. Given the following MPI code fragment that is sending messages from process with rank 0, complete the complementary receive operation to correctly receive the data being transmitted on process with rank 1 and return the information as a vector of strings. You must use only blocking MPI calls for receiving all the necessary information? [**3 points**]

```
void doManagerTasks(int argc, char *argv[]) {
  MPI_Send(&argc, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  for(int i = 0; (i < argc); i++) {
    std::string tmp = argv[i];
    MPI_Send(&tmp[0], tmp.size() + 1, MPI_CHAR, 1, 0,
             MPI_COMM_WORLD);
  }
  std::vector<std::string> ret;
  std::string rec;
  do {
    MPI_Recv(&rec[0], rec.size(), MPI_CHAR, 0, 1,MPI_COMM_WORLD);
    ret.push_back(rec);
  } while (rec.size > 0);
  return ret;
}
```

4. What is a deadlock? Explain using actual MPI code fragment that would experience a deadlock. [**2 points**]

   Deadlocks occur when two or more operations cannot complete due to some conflict (two send operations, or two receive operations with nothing to read, etc.)

```
// method 1
MPI_Send(&data, 1, MPI_INT, 1, INT_TAG, MPI_COMM_WORLD);
MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, NULL);

// method 2
MPI_Send(&data, 1, MPI_INT, 1, INT_TAG, MPI_COMM_WORLD);
MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, NULL);
```

   Both methods send and receive at the same time, so no message will actually be transmitted resulting in a deadlock.

5. List three suggestions for avoiding deadlocks. [**1 points**]

   a. Limit the size of messages being sent/received
   b. If large amounts of data must be sent, then break it up into fragments
   c. Always combine send and receive operations in the same loop

6. Illustrate an MPI program that has a clear load imbalance when it is run with 2 (or more) processes. [**2 points**]

```
void main(int argv, char* argc[]) {
    MPI_Init(&argv, &argc);
    int procs, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        sendTask();
    else
       doTask();

    MPI_Finalize();
}


// Only sends data to one process
void sendTask() {
    int msg;
    for (int i = 0; i < 100000; i++) {
       MPI_Send(&i, 1, MPI_INT, 1, 2, MPI_COMM_WORLD);
       MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, NULL);
    }
}

void doTask(int rank) {
    int master_rank = 0;
    int msg;
    MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, NULL);
    msg >> 4;
    MPI_Send(&msg, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
}
```

Due by:	**Tue, April 29 2016 BY 11:59 PM (EST)**