Week 10 Discussion Questions
Chapter 3.1 – 3.4:

I want you to design (no implementation yet) an MPI program to perform 2D Successive Over-Relaxation (SOR).

Recall that initially, the 2D-array $\text{va}_1$ contains 0.0s everywhere, except for the 1.0s down column 0. On each iteration, SOR updates all interior values (i.e., only the white values in the diagram change with the gray boundary values being fixed) by the average of their four nearest neighbors. Eventually after many iterations the values will stabilize. We won’t run to complete stabilization, but just until the maximum value change across the array during an iteration is less than a user specified threshold (e.g., 0.0001).

You’ll need to:

- decide how to decompose the work among MPI processes -- keep in mind load balancing and communication performance
- decide how to communicate among the MPI processes so all MPI processes have the data needed for the next iteration
- decide how to synchronize among the MPI processes so all MPI processes finish an iteration before any start the next iteration
- decide how to synchronize the MPI processes so all MPI processes stop if the max. change at any spot during an iteration is less than the threshold

Summarize your design by answering the following questions:

1) decide how to decompose the work among MPI processes -- keep in mind load balancing and communication performance

2) decide how to communicate among the MPI processes so all MPI processes have the data needed for the next iteration

3) decide how to synchronize among the MPI processes so all MPI processes finish an iteration before any start the next iteration

4) decide how to synchronize the MPI processes so all MPI processes stop if the max. change at any spot during an iteration is less than the threshold

5) How can we test and debug your program?
Blue Waters Supercomputer Usage

The Blue Waters supercomputer is accessible by logging on to your trail## account using Putty or ssh at: bwbay.ncsa.illinois.edu Your username and password is on the right-side of the Grade center on eLearning.

Compiling of a MPI program is done using the cc command (see below). Execution of a MPI program is done indirectly by submitting a “Portable Batch System (pbs) job” to the batch queue using the qsub command (see below) with a qsub file tailored for the MPI program. The qsub file contains the following PBS directives:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PBS -l nodes=4:ppn=2:xe</td>
<td>Specifies that 4 MPI nodes should be used to run the program with 2 MPI processes per node. Here a total of 8 MPI processes will be created when run. You can vary this to change the number of MPI processes, but you need to change the -n parameter of aprun to match!</td>
</tr>
<tr>
<td>#PBS -l walltime=00:05:00</td>
<td>Specifies the maximum cpu time the MPI program should be allocated.</td>
</tr>
<tr>
<td>aprun -n 16 ./greetings &gt; greeting_n2_ppn8_out.$PBS_JOBID</td>
<td></td>
</tr>
</tbody>
</table>

---

### Directions

- **mkdir directory greetings:**
  - `mkdir greetings`
  - `cd greetings`

- **Type in your MPI code using some editor (e.g., emacs):**
  - `emacs greetings.c`

- **Compile the MPI code using cc command:**
  - `cc -o greetings greetings.c`

- **Edit the qsub.greetings file to specify the number of nodes, processes per node (ppn), and aprun:**
  - `emacs qsub.greetings`

- **Submit qsub.greetings to the qsub queue:**
  - `qsub qsub.greetings`

- **List directory to see output files:**
  - `77#####.bw.out etc`
  - `ls`

- **Examine output files (use less (q to exit) or cat commands):**
  - `less *77*.bw.out`
  - `less *77*.bw.err`

- **View the status of all PBS jobs in queue:**
  - `qstat -u $(whoami)`

- **Delete a job from the PBS queue:**
  - `qdel 77#####.bw`

- **Copy lab/hw .zip from student.cs.uni.edu:**
  - `scp yourCatID@student.cs.uni.edu:lab10.zip .`

- **Unzip the .zip file:**
  - `unzip lab10.zip`

- **Zipping a folder (cd) to folder’s parent, then:**
  - `cd ..`
  - `zip lab10.zip lab10/*`

### Commands

You can scp from Blue Waters login computer back to student.cs.uni.edu:

`scp lab10.zip yourCatID@student.cs.uni.edu:lab10.zip`

**NOTE:** You’ll need to supply your CatID password to complete the copy

You can scp to Blue Waters login computer from student.cs.uni.edu:

`scp yourCatID@student.cs.uni.edu:xyz.zip xyz.zip`

**NOTE:** You’ll need to supply your CatID password to complete the copy

Blue Waters page 1
Blue Waters Supercomputer Usage

My gsub.greetings file in my greetings subdirectory.

#!/bin/bash
# NOTE: lines that begin with "#PBS" are not interpreted by the shell but ARE used by the
# batch system, whereas lines that begin with multiple # signs, like "###PBS" are
# considered "commented out" by the batch system and have no effect.

### set the number of nodes and set the number of PEs per node
#PBS -l nodes=2:ppn=8:xe
### set the wallclock time
#PBS -l walltime=00:05:00
### set the job name
#PBS -N greetings
### set the job stdout and stderr
#PBS -e greetings_$PBS_JOBID.err
#PBS -o greetings_n2_ppn8_$PBS_JOBID.out

# to launch a job in a directory prepared for the job to run within, you'll want to cd to
# that directory
cd $PBS_O_WORKDIR

# Alternatively, the job script can create its own job-ID-unique directory to run within.
# In that case you'll need to create and populate that directory with executables and
# perhaps inputs
# [uncomment and customize the following lines to enable this behavior]
# mkdir -p /scratch/sciteam/$USER/$PBS_JOBID
# cd /scratch/sciteam/$USER/$PBS_JOBID
# cp /scratch/job/setup/directory/* .

# To add certain modules that you do not have added via ~/.modules
# /opt/modules/default/init/bash # NEEDED to add module commands to shell
#module load craype-hugepages2M perftools

# export APRUN_XFSR_LIMITS=1 # to transfer shell limits to the executable
### launch the application
### redirecting stdin and stdout if needed
### NOTE: (the file must exist for redirected stdin input)

aprun -n 16 ./greetings > greetings_n2_ppn8_out.$PBS_JOBID
### For more information see the man page for aprun
instr006@h2ologin2:~/greetings> qsub qsub.greetings
INFO: Job submitted to account: baoe
7718152.bw
instr006@h2ologin2:~/greetings> qstat -u $(whoami)

hwsched.ncsa.illinois.edu: Blue_Waters

<table>
<thead>
<tr>
<th>Job ID</th>
<th>Username</th>
<th>Queue</th>
<th>Jobname</th>
<th>SessID</th>
<th>NDS</th>
<th>TSK</th>
<th>Req'd Memory</th>
<th>Req'd Time</th>
<th>S</th>
<th>Elap Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>7718152.bw</td>
<td>instr006</td>
<td>normal</td>
<td>greetings</td>
<td>--</td>
<td>2</td>
<td>16</td>
<td>--</td>
<td>00:05:00</td>
<td>0</td>
<td>--</td>
</tr>
</tbody>
</table>

instr006@h2ologin2:~/greetings> ls

greetings

greetings_7718152.bw.err

greetings_7718152.bw.out

greetings.c

qsub.greetings

instr006@h2ologin2:~/greetings> cat greetings_n2_ppn_8_7718152.bw.out

Begin Torque Prologue on nid25354
at Tue Oct 24 10:14:57 CDT 2017
Job Id: 7718152.bw
Username: instr006
Group: TRAIN_baoe
Job name: greetings
Requested resources: neednodes=2:ppn=8:xe, nodes=2:ppn=8:xe, walltime=00:05:00
Queue: normal
Account: baoe
End Torque Prologue: 0.078 elapsed

instr006@h2ologin2:~/greetings> cat greetings_n2_ppn8_out.7718152.bw

Greetings from process 4 on nid21985!
Greetings from process 5 on nid21985!
Greetings from process 6 on nid21985!
Greetings from process 7 on nid21985!
Greetings from process 8 on nid21985!
Greetings from process 9 on nid21985!
Greetings from process 10 on nid21985!
Greetings from process 11 on nid21985!
Greetings from process 12 on nid01295!
Greetings from process 13 on nid01295!
Greetings from process 14 on nid01295!
Greetings from process 15 on nid01295!

Application 62870565 resources: utime ~2s, stime ~6s, Res ~22636, inblocks ~17715, outblocks ~36640

instr006@h2ologin2:~/greetings>
/ * FILE: greetings.c -- greetings program
* Compile by: cc -o greetings greetings.c
* Run by: gsub gsub.greetings
* 
* Send a message from all processes with rank != 0 to process 0.
* Process 0 prints the messages received.
* 
* Input: none.
* Output: contents of messages received by process 0.
*/

#include <stdio.h>
#include <string.h>
#include <mpi.h>

main(int argc, char* argv[]) {
    int my_rank; /* rank of process */
    int p; /* number of processes */
    int source; /* rank of sender */
    int dest; /* rank of receiver */
    int tag = 0; /* tag for messages */
    int length; /* size of the hostname */
    char message[100]; /* storage for message */
    char name[100]; /* storage for the host */
    MPI_Status status; /* return status for receive */
    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        /* Create message */
        MPI_Get_processor_name(name, &length);
        sprintf(message, "Greetings from process %d on %s!", my_rank, name);
        dest = 0;
        /* Use strlen+1 so that \0 gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        } // end for
    } // end if

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
Learning Objectives:
- Apply MPI library commands to create simple MPI processes that communicate by MPI_Send and MPI_Recv
- Compile and execute MPI programs using qsub commands on Blue Waters supercomputer
- Analyze MPI performance

To start the lab:
- watch the Lab 10 Video on the eLearning system
- download lab10.zip file from eLearning to your computer
- upload lab10.zip file from your computer to your account on student.cs.uni.edu
- log on to your tral# Blue Water account by pointing a ssh client (e.g., Putty) at to: bwbay.ncsa.illinois.edu
- The first access will require a few questions to agree to Blue Waters Terms of Use Policy. On a subsequent login session you will bounce through bwbay to one of three login nodes.
- “secure” copy lab10.zip from student.cs.uni.edu to your tral# Blue Water account using scp:
  (NOTE the ‘.’ at the end of the command is needed and denotes the current directory. You’ll need to supply your student.cs.uni.edu CatID password to complete the copy):
  scp YOUR_STUDENT-CS.UNI.EDU_USERNAME@student.cs.uni.edu:lab10.zip
- Unzip lab10.zip on Blue Waters by:
  unzip lab10.zip

Part A: Use an editor to open the file sum1DArrayA.c which contains an MPI program to sum a 1D array using multiple MPI processes.

a) The RootProcess (myID of 0) creates a 1-dimensional array whose size is specified by a command-line argument and fills the array with random floating-point numbers. How do the other MPI processes get their block of the 1D array to sum?

b) After each MPI process calculated their localSum, how is the sum of the whole array calculated?

Compile (cc -o sum1DArrayA -O3 sum1DArrayA.c) and run by: qsub qsub.sum1DArrayA
which starts the program with a command-line array size of 1024 by the aprun command of:
aprun -n 8 -N 2 ./sum1DArrayA 1024 > sum1DArrayA_1024_out.$PBS_JOBID
It might take a while before your queued program runs. You can monitor the program by: qstat -u ${whoami}
c) Why is the sequential sum calculation by only the RootProcess faster than the parallel sum calculation?

d) If we had a large 1D array, would you expect the parallel sum calculation to be faster than the sequential sum calculation? (You can change the array size from 1024 to 1048576 by selecting the alternate aprun command-line argument in the qsub.sum1DArrayA file and rerun)
Part B: Using an editor open the file sum1DArrayB.c which contains another MPI program to sum a 1D array using multiple MPI processes.

a) The RootProcess (myID of 0) creates a 1-dimensional array whose size is specified by a command-line argument and fills the array with random floating-point numbers. How do the other MPI processes get their block of the 1D array to sum?

b) After each MPI process calculated their localSum, how is the sum of the whole array calculated?

Compile (cc -o sum1DArrayB -O3 sum1DArrayB.c) and run by: qsub qsub.sum1DArrayB

c) Why is the parallel sum calculation of sum1DArrayB.c faster than sum1DArrayA.c?

d) Change the array size in the aprun command-line argument in the qsub.sum1DArrayB to 1025 and rerun the program. Why is the parallel sum calculation incorrect (i.e., does match the sequential sum)?

Part C: Copy sum1DArrayB.c to sum1DArrayC.c. Edit the sum1DArrayC.c to distribute the 1D array to MPI processes by MPI_Scatterv. See https://www.cac.cornell.edu/vw/MPIcc/gatherscatter.aspx for information about MPI_Scatterv. Once you get your sum1DArrayC.c program working, copy your lab10 folder to student.cs.uni.edu by:
cd ...
zip lab10.zip lab10/*
scp lab10.zip yourCatIDname@student.cs.uni.edu:lab10.zip
(NOTE: you’ll need to supply your CatID password to complete the copy)

You can transfer lab10.zip to your local computer using FileZilla/WinSCP/etc. and unzip it. Add your answers to these questions and re-zip it for eLearning submission.

Submit lab10.zip containing question answers and completed program on the eLearning system
1. Chapter 3 deals with distributed-memory programming with MPI (Message Passing Interface). MPI is a distributed memory programming model in which a collection of processes communicate by sending messages.

```c
/* File: sum1DArray.c
 * Compile as: mpicc -o sum1DArray sum1DArrayA.c
 * Run by: qsub qsub sum1DArrayA
 * Description: An MPI solution to sum a 1D array. */

#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <stdio.h>
#include <mpi.h>
#include "timer.h"

#define RootProcess 0

const int tag = 1;

int main(int argc, char* argv[]) {
    int myID, value, numProcs, i, p;
    float * myArray;
    double seqSum, parallelSum, localSum, recvSum;
    int length;
    int length_per_process;
    double clockStart, clockEnd;

    GET_TIME(clockStart);

    MPI_Status status;

    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs); /* Get rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &myID); /* Get rank */

    // all processes have access to argc and argv
    sscanf(argv[1], "%d", &length);
    length_per_process = length/numProcs;

    if ( myID == RootProcess ) {
        if (argc != 2) {
            printf("Usage: %s <# of floats to sum>\n", argv[0]);
            return(0);
        }

        // Generate data array  length = SIZE;
        printf("length = %d\n", length);
        myArray=(float *) malloc(length*sizeof(float));
        srand(5);
        for (i=0; i < length; i++) {
            myArray[i] = rand() / (float) RAND_MAX;
        } // end for i
    }

    GET_TIME(clockEnd);

    MPI_Finalize(); /* Finalize MPI */

    return(0);
}
```
/* Send a message with part of array to each MPI process */
for (p=0; p<numProcs-1; p++) {
    MPI_Send( myArray+length_per_process*p, length_per_process, MPI_FLOAT, p+1, tag,
              MPI_COMM_WORLD);
} // end for p

/* Do the actual work - sum "right" end of array*/
localSum = 0.0;
for (i=length_per_process*(numProcs-1); i<length; i++) {
    localSum += myArray[i];
} // end for i

parallelSum = localSum;
/* collect up localSum's from all other processes */
for (p=0; p<numProcs-1; p++) {
    MPI_Recv( &recvSum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status );
    parallelSum += recvSum;
} // end for

} else { // code for MPI processes that are not the root process
myArray = (float *) malloc(length_per_process*sizeof(float));
MPI_Recv( myArray, length_per_process, MPI_FLOAT, RootProcess, tag, MPI_COMM_WORLD,
          &status );

/* Do the actual work */
localSum = 0.0;
for (i=0; i<length_per_process; i++) {
    localSum += myArray[i];
} // end for i

MPI_Send( &localSum, 1, MPI_DOUBLE, RootProcess, tag, MPI_COMM_WORLD );
}

} // end if

GET_TIME(clockEnd);

if (myID == RootProcess) {
    printf( "Time to sum %d floats with MPI in parallel %.3f seconds\n", length,
            (clockEnd - clockStart));
    GET_TIME(clockStart);
    seqSum = 0.0;
    for (i=0; i<length; i++) {
        seqSum += myArray[i];
    } // end for i
    GET_TIME(clockEnd);
    printf( "Time to sum %d floats sequentially %.3f seconds\n", length,
            (clockEnd - clockStart));
    printf("The parallel sum: %f\n", parallelSum);
    printf("The sequential sum: %f\n", seqSum);
}

free(myArray);

MPI_Finalize();
return 0;
} /* end main */
Comp. Arch.

**Week 10 Monday**

**MPI_Init**

```c
int MPI_Init(int *argc, char **argv) {
    /* Initialize MPI */
    /* Number of command line arguments */
    /* Command line arguments */
    return MPI_SUCCESS;
}
```

**MPI_Finalize**

```c
int MPI_Finalize() {
    /* Finalize MPI */
    return MPI_SUCCESS;
}
```

**Notes:**
- This routine should be the last MPI routine called in each process, and it should only be invoked after all other MPI routines have completed. In particular, any pending communication operations should complete before this routine is called.
- Return values:
  - An MPI success code.

**MPI_Comm_Size**

```c
int MPI_Comm_Size(int *size, int *rank) {
    /* Retrieve the number of tasks in the specified communicator */
    /* The number of tasks */
    return MPI_SUCCESS;
}
```

**Notes:**
- This routine retrieves a process' rank within a communicator.
- Return values:
  - An MPI success code.

**MPI_Init**

```c
int MPI_Init(int *argc, char **argv) {
    /* Initialize MPI */
    /* Number of command line arguments */
    /* Command line arguments */
    return MPI_SUCCESS;
}
```

**MPI_Finalize**

```c
int MPI_Finalize() {
    /* Finalize MPI */
    return MPI_SUCCESS;
}
```

**Notes:**
- This routine should be the last MPI routine called in each process, and it should only be invoked after all other MPI routines have completed. In particular, any pending communication operations should complete before this routine is called.
- Return values:
  - An MPI success code.

**MPI_Comm_Size**

```c
int MPI_Comm_Size(int *size, int *rank) {
    /* Retrieve the number of tasks in the specified communicator */
    /* The number of tasks */
    return MPI_SUCCESS;
}
```

**Notes:**
- This routine retrieves a process' rank within a communicator.
- Return values:
  - An MPI success code.

**MPI_Init**

```c
int MPI_Init(int *argc, char **argv) {
    /* Initialize MPI */
    /* Number of command line arguments */
    /* Command line arguments */
    return MPI_SUCCESS;
}
```

**MPI_Finalize**

```c
int MPI_Finalize() {
    /* Finalize MPI */
    return MPI_SUCCESS;
}
```

**Notes:**
- This routine should be the last MPI routine called in each process, and it should only be invoked after all other MPI routines have completed. In particular, any pending communication operations should complete before this routine is called.
- Return values:
  - An MPI success code.
Comp. Arch.

Week 10 Monday

MPI Scatter could be used to send equal size blocks to each process

```c
MPI_Scatter(
int MPI_Scatter(
    void *sendbuffer,       // Address of the data to send
    int sendcount,          // Number of data elements to send
    MPI_Datatype sendtype,  // Type of data elements to send
    int destbuffer,         // Address of buffer to receive data
    int destcount,          // Number of data elements to receive
    MPI_Datatype desttype,  // Type of data elements to receive
    int root,               // Rank of the root process
    MPI_Comm *comm          // An MPI communicator
)
)
```

Arguments:

- The first three arguments specify the address, size, and type of the data elements to send to each process. These arguments only have meaning for the root process.
- The second three arguments specify the address, size, and type of the data elements for each receiving process. The size and type of the sending data and the receiving data may differ as a means of converting data types.
- The seventh argument specifies the root process that is the source of the data.
- The eighth argument specifies the MPI communicator to use.

Notes:

This routine distributes data from the root process to all other processes, including the root. A more sophisticated version of the routine, `MPI_Scatterv()`, allows the root process to send different amounts of data to the various processes. Details can be found in the MPI standard.

Return value:

An MPI error code.

MPI_Scatterv and MPI_Gatherv should be used if each process does not receive the same number of data items.

```c
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void *
recvbuf, int recvcounts, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN sendbuf address of send buffer (choice, significant only at root)
IN sendcounts non-negative integer array (of length group size) specifying the number of elements to send to each processor
IN displs integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcount number of elements in receive buffer (non-negative integer)
IN recvtype data type of receive buffer elements (handle)
IN rootrank of sending process (integer)
IN comm communicator (handle)
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <stdio.h>
#include <mpi.h>
#include "timer.h"

#define RootProcess 0

const int tag = 1;

int main(int argc, char* argv[]) {
    int myID, value, numProcs, i, p;
    float *myArray;
    double seqSum, parallelSum, localSum, recvSum;
    int length;
    int length_per_process;
    double clockStart, clockEnd;

    GET_TIME(clockStart);

    MPI_Status status;

    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs); /* Get rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &myID); /* Get rank */

    // all processes have access to argc and argv
    scanf(argv[1], "%d", &length);
    length_per_process = length/numProcs;

    if (myID == RootProcess) {
        if (argc != 2) {
            printf("Usage: %s <# of floats to sum>\n", argv[0]);
            return(0);
        }

        // Generate data array
        length = SIZE;
        printf("Length = %d\n", length);
        myArray = (float *) malloc(length*sizeof(float));
        srand(5);
        for (i=0; i < length; i++) {
            myArray[i] = rand() / (float) RAND_MAX;
        } // end for i

        // Send a message with part of array to each MPI process
        for (p=0; p<numProcs-1; p++) {
            MPI_Send( myArray, length_per_process*p, length_per_process, MPI_FLOAT,
                p+1, tag, MPI_COMM_WORLD );
        } // end for p

        /* Do the actual work - sum "right" end of array*/
        localSum = 0.0;
    }
```c
for (i = length_per_process*(numProcs-1); i < length; i++) {
    localSum += myArray[i];
} // end for i

parallelSum = localSum;
/* collect up localSum's from all other processes */
for (p = 0; p < numProcs-1; p++) {
    MPI_Recv(&recvSum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,
    &status);
    parallelSum += recvSum;
} // end for

} else { // code for MPI processes that are not the root process
    myArray = (float *) malloc(length_per_process*sizeof(float));
    MPI_Recv(myArray, length_per_process, MPI_FLOAT, RootProcess, tag, MPI_COMM_WORLD,
    &status);

    /* Do the actual work */
    localSum = 0.0;
    for (i = 0; i < length_per_process; i++) {
        localSum += myArray[i];
    } // end for i

    MPI_Send(&localSum, 1, MPI_DOUBLE, RootProcess, tag, MPI_COMM_WORLD);

} // end if

GET_TIME(clockEnd);

if (myID == RootProcess) {
    printf("Time to sum %d floats with MPI in parallel %3.5f seconds\n", length,
    (clockEnd - clockStart));
    GET_TIME(clockStart);
    seqSum = 0.0;
    for (i = 0; i < length; i++) {
        seqSum += myArray[i];
    } // end for i
    GET_TIME(clockEnd);
    printf("Time to sum %d floats sequentially %3.5f seconds\n", length,
    (clockEnd - clockStart));

    printf("The parallel sum: %f\n", parallelSum);
    printf("The sequential sum: %f\n", seqSum);

} // end if
free(myArray);

MPI_Finalize();
return 0;
} /* end main */
```
Comp. Arch.

1. MPI_Send and MPI_Recv are blocking, but exact behavior depends on the implementation. Options:
   - Copy to MPI “Kernel” buffer and return “immediately”, or
   - Block until sent as a result of a matching MPI_Recv.

All MPI implementations guarantee that messages sent by a process to other processes are non-overtaking (i.e., arrive in the order sent). However, with multiple processes running independently and performing blocking MPI_Send and MPI_Recv we can be nondeterministic behaviour.

a) If the first option above is chosen, what problem might occur if the process continues to send faster than the corresponding receives are performed?

b) Due to the blocking nature of these functions, what must we be careful to avoid?

c) Either way latency for MPI_Send and MPI_Recv can be high. Besides the network latency, the sent message is copied from the program’s “buffer” to the kernel address space.

d) How might non-blocking MPI_Isend and MPI_Irecv be used to hide communication latency? (Note: MPI_Wait can be used to can block until the specified operation (i.e., MPI_Isend) has completed globally, and MPI_Test returns a boolean indicating the status of a specified operation)

Other Communication Modes:
- Synchronous Send (MPI_Ssend() and MPI_Issend()) are like a rendezvous in Ada, i.e., the sender blocks until the receiving process begins to receive the message.
- Buffered Send (MPI_Bsend() and MPI_IBsend()) - process provides the buffer space to send from to avoid insufficient kernel buffer space. MPI_Buffer_attach() and MPI_Buffer_detach() routines should be used to specify the allocated memory.
- Ready Send (MPI_Rsend and MPI_Irsend) allow a message to be placed directly into a memory location avoiding handshaking and buffering costs. Warning: Programmer must guarantee that the Receive operation has been initiated before the message arrives. If the timing assumption is violated, an error will be flagged when the Receive operation is performed.
2. Collective Communication - high-level communication operations that involve a group of MPI processes. For example:

- scatter and gather routines
- reduce and scan routines
- broadcast routines
- barrier routines

An MPI process can belong to multiple groups. Each process in the group is assigned a unique rank (ID #) between 0 to (group size)-1. If we are thinking of the processes in a 2D, each row of processes could be in their own group or each column of processes could be in their own group.

Once we have a group, we can create a communicator to allow collective communications with all group members.

```
MPI_Comm_create()
int MPI_Comm_create(
MPI_Comm comm,
MPI_Group group,
MPI_Group newGroup,
MPI_Group *newComm)

Notes:
Creates a new communicator for a given group.
Return value:
An MPI error code.
```

If we are thinking of the processes in a 2D grid and want each row of processes to perform collective communication (i.e., broadcast cross row of processes), The below figure gives the rough idea of how.

```c
int numCols;         /* initialized elsewhere */

void broadcast_example()
{
    int **ranks;    /* the ranks that belong to each group */
    int myRank;
    int rowNumber;  /* row number of this process */
    int random;     /* value that we would like to broadcast */
    rowNumber = myRank / numCols;

    MPI_Group globalGroup, newGroup;
    MPI_Comm zoneComm(numCols);

    /* Initialize ranks[][] array */
    ranks[0] = {0, 1, 2, 3}; // not legal C */
    ranks[1] = {4, 5, 6, 7};
    ranks[2] = {8, 9, 10, 11};
    ranks[3] = {12, 13, 14, 15};

    /* Extract the original group handle */
    MPI_Comm_group(MPI_COMM_WORLD, &globalGroup);

    /* Define the new group */
    MPI_Group_incl(globalGroup, 4, numCols, ranks[rowNumber], &newGroup);

    /* Create new communicator */
    MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);

    random = rand();

    /* Broadcast 'random' across rows */
    MPI_Bcast(&random, 1, MPI_CHAR, rowNumber, numCols, zoneComm);
}
```
3. Reduction (MPI_Reduce and MPI_Allreduce) operations:

<table>
<thead>
<tr>
<th>Operation Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_EXOR</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of minimum</td>
</tr>
</tbody>
</table>

a) How could a broadcast be implemented?

b) How does this differ from the Scatter and Gather communications?

c) Why might it be useful (i.e., faster to bundle several smaller messages into one instead of sending multiple smaller messages?)
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <stdio.h>
#include <mpi.h>
#include "timer.h"

#define RootProcess 0

const int tag = 1;

int main(int argc, char* argv[]) {
    int myID, value, numProcs, i, p;
    float * myArray;
    float * localArray;
    double seqSum, parallelSum, localSum, recvSum;
    int * counts;
    int * displacements;
    int length;
    int length_per_process;
    double clockStart, clockEnd;
    
    GET_TIME(clockStart);

    MPI_Status status;

    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs); /* Get rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &myID); /* Get rank */

    // all processes have access to argc and argv
    sscanf(argv[1], "%d", &length);
    length_per_process = length/numProcs;

    if (myID == RootProcess) {
        if (argc != 2) {
            printf("Usage: %s # of floats to sum>\n", argv[0]);
            return(0);
        }

        // Generate data array length = SIZE;
        printf("length = %d\n", length);
        myArray=(float*) malloc(length*sizeof(float));
        srand(5);
        for (i=0; i < length; i++) {
            myArray[i] = rand() / (float) RAND_MAX;
        }
    } // end if (myID == RootProcess
// code for all MPI processes.

localArray = (float *) realloc(length_per_process * sizeof(float));
printf("myID: %d\n", myID);
// distributes same sized chunks of data to all processes
MPI_Scatter(myArray, length_per_process, MPI_FLOAT, localArray, length_per_process, MPI_FLOAT, RootProcess, MPI_COMM_WORLD);

/* Do the actual work */
localSum = 0.0;
for (i=0; i < length_per_process; i++) {
    localSum += localArray[i];
} // end for i

MPI_Reduce(&localSum, &parallelSum, 1, MPI_DOUBLE, MPI_SUM, RootProcess, MPI_COMM_WORLD);

GET_TIME(clockEnd);

if (myID == RootProcess) {
    printf("Time to sum %d floats with MPI in parallel $3.5f$ seconds\n", length, (clockEnd - clockStart));
    GET_TIME(clockStart);
    seqSum = 0.0;
    for (i=0; i < length; i++) {
        seqSum += myArray[i];
    } // end for i
    GET_TIME(clockEnd);
    printf("Time to sum %d floats sequentially $3.5f$ seconds\n", length, (clockEnd - clockStart));
    printf("The parallel sum: $%.f$\n", parallelSum);
    printf("The sequential sum: $%.f$\n", seqSum);
    free(myArray);
} // end if
free(localArray);

MPI_Finalize();
return 0;
} /* end main */
Week 12 Discussion Questions

Learning Objectives:
- Analyze larger MPI programs for key design decisions, and sources of parallel overhead.
- Analyze MPI performance.

MPI versions of Chapter 6 “real-world” examples: n-body problem and TSP (traveling-salesperson problem).

Serial code:

```c
Get input data;
for each timestep {
    if (timestep output) Print positions and velocities of each particle q;
    for each particle q
        Compute total force on q;
    for each particle q
        Compute position and velocity of q;
}
Print positions and velocities of particles;
```

The for each particle q:

```
0  f_{01}  f_{02}  ...  f_{0,n-1}
-f_{01}   0  f_{12}  ...  f_{1,n-1}
-f_{02}  -f_{12}   0  ...  f_{2,n-1}
  .   .   .   .   .
-f_{0,n-1}  -f_{1,n-1}  -f_{2,n-1}  ...  0
```

“Computer total force on q” code needs to perform individual force calculations where:

- row 0 are the forces on particle 0 by other particles, etc.
- matrix is “symmetric”, except opposite forces are negated (two versions: basic and reduced utilizing symmetry)

1. In the MPI version we could assign an equal number of particles to each process(or) with:
   - Each process stores the entire global array of particle masses.
   - Each process only uses a single n-element array for the positions.
   - Each process uses a pointer local_pos that refers to the start of its block of pos.
   - So on process 0 local_pos = pos; on process 1 local_pos = pos + loc_n; etc.

MPI algorithm for basic n-body problem:

```c
Get input data;
for each timestep {
    if (timestep output)
        Print positions and velocities of particles;
    for each local particle loc_q
        Compute total force on loc_q;
    for each local particle loc_q
        Compute position and velocity of loc_q;
    Allgather local positions into global pos array;
}
Print positions and velocities of particles;
```

```c
int MPI_Allgather(
    void* send_buf_p /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    MPI_Comm comm /* in */;
```

NOTES on MPI_Allgather:

Concatenates the contents of each process’ send_buf_p and stores this in each process’ recv_buf_p.
- recv_count is the amount of data being received

a) Why is MPI_Allgather only used to distribute the positions of all particles, but not their velocities?
b) How are processes synchronized for each timestep?

2. As with pthreads we can statically allocate the work to each MPI process by having process 0 do a breadth-first search (BFS) of "state-space tree" until it has enough tree nodes for all the processes.

![Diagram of BFS result by process 0]

a) What MPI routine could we use to distribute the tree nodes to all processes?

b) Would each process get the same number of tree nodes?

c) Each process could run independently to completion using their local best tour for pruning, but why is maintaining a "global" best tour better?

d) What is wrong with each of the following ways to try to maintain a global best tour?
   - Using MPI_Bcast to broadcast new best tour found by a process
   - Looping to MPI_Send a "new best tour message" to all other processes individually with each process periodically performing a MPI_Recv of "new best tour messages"

c) A process can use the non-blocking MPI_Iprobe routine to check to see if a message is available, but which might be the source process?

```c
int MPI_Iprobe(
    int source, /* in */,
    int tag, /* in */,
    MPI_Comm comm, /* in */,
    int* msg_avail_p /* out */,
    MPI_Status* status_p /* out */);
```

f) How might we use the tag parameter?
Week 12 Discussion Questions

4. If a process runs out of work (completed searching its assigned subtree(s)), what should it do?

5. If process 0 is out of work and received a “completion message” from everybody, how can it determine the global best tour?

6. As with pthreads, the MPI processes can dynamically allocate the tree search by allowing an MPI process which runs out of work to obtain work from another process. Outline the procedure for processes to dynamically request and receive work from another process.

7. How can we detect that all processes have runs out of work?
Learning Objectives:
- Analyze larger MPI programs for key design decisions, and sources of parallel overhead.
- Estimate the run-time of larger problems from timings of smaller instances using big-oh notation.

To start the lab:
- re-read chapter 6 in the textbook
- watch the Lab 12 Video on the eLearning system
- download lab12.zip file from eLearning to your computer
- upload lab12.zip file from your computer to your account on student.cs.uni.edu
- log on to your tral## Blue Water account by pointing a ssh client (e.g., Putty) at to:
  bwbay.ncsa.illinois.edu
- “secure” copy lab12.zip from student.cs.uni.edu to your tral## Blue Water account using scp:
  (NOTE the ‘.’ at the end of the command is needed and denotes the current directory. You’ll need to supply
  your student.cs.uni.edu CatID password to complete the copy):
  scp YOUR_STUDENT.CS.UNI.EDU_USERNAME@student.cs.uni.edu:lab12.zip .
- Unzip lab12.zip on Blue Waters by:
  unzip lab12.zip

Part A: Make a copy of lab12/mpi_tsp_stat.c and modify it as described in Exercise 6.21 (a) – on
back-side. Write a report comparing performance with the original mpi_tsp_stat.c program.

EXTRA CREDIT Part B: Using an editor on Blue Waters open the file lab12/mpi_nbody_basic.c
which contains an MPI program for the “basic” n-body problem. Copy and modify this program as described in
Exercise 6.12 (a). Write a report comparing performance with the original n-body solver.

EXTRA CREDIT Part C: Make another copy of lab12/mpi_nbody_basic.c and modify it as described
in Exercise 6.12 (b). Write a report comparing performance with the original n-body solver.

Submit lab12.zip containing modified programs and performance comparisons on the eLearning system
6.12. a. Modify the basic MPI implementation of the \( n \)-body solver so that it uses a separate array for the local positions. How does its performance compare with the performance of the original \( n \)-body solver? (Look at performance with I/O turned off.)

b. Modify the basic MPI implementation of the \( n \)-body solver so that it distributes the masses. What changes need to be made to the communications in the program? How does the performance compare with the original solver?

6.21. a. Modify the static MPI TSP program so that each process uses a local best tour data structure until it has finished searching. When all the processes have finished executing, the processes should execute a global reduction to find the least-cost tour. How does the performance of this implementation compare to the static implementation? Can you find input problems for which its performance is competitive with the original static implementation?

b. Create a TSP digraph in which the initial tours assigned to processes \( 1,2,\ldots,\text{comm.sz} - 1 \) all have an edge that has a cost that is much greater than the total cost of any complete tour that will be examined by process 0. How do the various implementations perform on this problem when \( \text{comm.sz} \) processes are used?
Learning Objectives:
- Design an efficient data-decomposition (i.e., block vs. cyclic) for a MPI program in C.
- Write correct C program using MPI library commands to initialize (create) MPI processes, communicate data, and synchronize their operation.
- Time various sizes and # of MPI processes

Homework #11 Description:
You are to design and write a C program utilizing MPI to efficiently perform 2D Successive Over-Relaxation (SOR) problem. To start the homework, download and extract hw9.zip from the eLearning system. It contains a "starter" program hw9/hw9.c which you can use as your starting point.

Recall that initially, the 2D-array va contains 0.0s everywhere, except for the 1.0s down column 0. On each iteration, SOR updates all interior values (i.e., only the white values in the diagram change with the gray boundary values being fixed) by the average of their four nearest neighbors. Eventually after many iterations the values will stabilize. We won’t run to complete stabilization, but just until the maximum value change across the array during an iteration is less than a user specified threshold (e.g., 0.0001).

You’ll need to:
- decide how to decompose the work among MPI processes -- keep in mind load balancing and communication performance
- decide how to communicate among the MPI processes so all MPI processes have the data needed for the next iteration
- decide how to synchronize among the MPI processes so all MPI processes finish an iteration before any start the next iteration
- decide how to synchronize the MPI processes so all MPI processes stop if the max.
  change at any spot during an iteration is less than the threshold
- complete the perform2D_SOR function run by all the MPI processes
- test and debug your program

1) Download the starter code hw9.zip which is available on the eLearning system
2) For this activity I want you to:
   - use FileZilla, WinSCP, scp, ... to copy the starter code hw9.zip to student.cs.uni.edu
   - log-on you Blue Waters account
   - use scp YOUR_STUDENT.CS.UNI.EDU_USERNAME:hw9.zip 
   - use an editor (emacs or nano) to complete the hw9.c program
   - compile the C to an MPI executable file using: cc -o hw9 hw9.c
   - edit the qsub.hw9 file to vary the number of processes and matrix sizes for table below

3) Use a secure ftp client (e.g., FileZilla, WinSCP, scp, etc.) to copy your hw11 directory back to your local computer
   (On a MAC you can probably use: scp -r userName@briareus.physics.uni.edu:/hw11 localDir)

4) On your local computer zip the hw11 directory and submit in on the eLearning system

Complete the following table for your program by varying the qsub.hw9 commands:
#PBS -l nodes=4:ppn=2:xe
aprun -n 8 ./hw9 1024 0.0001 > hw9_1024_out.$PBS_JOBID
This means use 4 nodes with 2 processes per node for a total of 8 MPI processes NOTE: The aprun -n parameter must match the total number of processes, e.g., 8 in this example
### Homework #9

**Due: Wednesday, Nov. 20 at 5 PM (2 weeks)**

<table>
<thead>
<tr>
<th>qsub command</th>
<th># MPI processes</th>
<th>Time of sequential 2D SOR (in seconds)</th>
<th>Time of your parallel 2D SOR (in seconds) using a varying matrix sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#PBS -l nodes=4:ppn=1:xe</code></td>
<td>4</td>
<td></td>
<td>1024 x 1024</td>
</tr>
<tr>
<td><code>#PBS -l nodes=2:ppn=2:xe</code></td>
<td>4</td>
<td></td>
<td>2500 x 2500</td>
</tr>
<tr>
<td><code>#PBS -l nodes=1:ppn=4:xe</code></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#PBS -l nodes=4:ppn=2:xe</code></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#PBS -l nodes=2:ppn=4:xe</code></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#PBS -l nodes=3:ppn=2:xe</code></td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#PBS -l nodes=4:ppn=4:xe</code></td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#PBS -l nodes=2:ppn=8:xe</code></td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explain your timing results.**

Submit hw9.zip containing a completed timing table with explanation and completed program (hw9.c) on the eLearning system.
/* File: mpi_nbody_basic.c
 * Purpose: Implement a 2-dimensional n-body solver that uses the
 * basic algorithm. This version uses an in-place Allgather
 * Compile: mpicc -g -Wall -o mpi_nbody_basic mpi_nbody_basic.c -lm
 * To turn off output (e.g., when timing), define NO_OUTPUT
 * To get verbose output, define DEBUG
 * Run: mpiexec -n <number of processes> ./mpi_nbody_basic
 * <number of particles> <number of timesteps> <size of timestep
 >
 * <output frequency> <g|i>
 * 'g': generate initial conditions using a random number
 * generator
 * 'i': read initial conditions from stdin
 * number of particles should be evenly divisible by the number
 * of MPI processes
 * A stepsize of 0.01 seems to work well with automatically
 * generated data.
 *
 * Input: If 'g' is specified on the command line, none.
 * If 'i', mass, initial position and initial velocity of
 * each particle.
 * Output: If the output frequency is k, then position and velocity of
 * each particle at every kth timestep. This value is
 * ignored (but still necessary) if NO_OUTPUT is defined
 *
 * for each timestep t {
 * for each particle i I own
 * compute F(i), the total force on i
 * for each particle i I own
 * update position and velocity of i using F(i) = ma
 * Allgather positions
 * if (output step) {
 * Allgather velocities
 * Output new positions and velocities
 * }
 * }
 *
 * Force: The force on particle i due to particle k is given by
 * -G m_i m_k (s_i - s_k)/|s_i - s_k|^3
 *
 * Here, m_j is the mass of particle j, s_j is its position vector
 * (at time t), and G is the gravitational constant (see below).
 * Note that the force on particle k due to particle i is
 * -(force on i due to k). So we could approximately halve the number
 * of force computations. This version of the program does not
 * exploit this.
 *
 * Integration: We use Euler's method:
 * v_i(t+1) = v_i(t) + h v'_i(t)
 * s_i(t+1) = s_i(t) + h v_i(t)
 *
 * Here, v_i(t) is the velocity of the ith particle at time t and
 * s_i(t) is its position.
* * Notes: * * 1. Each process stores the masses of all the particles: the * masses array has dimension n = number of particles. * * IPP: Section 6.1.9 (pp. 290 and ff.) */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h>

#define DIM 2 /* Two-dimensional system */
#define X 0    /* x-coordinate subscript */
#define Y 1    /* y-coordinate subscript */

typedef double vect_t[DIM]; /* Vector type for position, etc. */

/* Global variables. Except or vel all are unchanged after being set */
const double G = 6.673e-11; /* Gravitational constant. */
    /* Units are m^3/(kg*s^2) */

int my_rank, comm_sz;
MPI_Comm comm;
MPI_Datatype vect_mpi_t;

/* Scratch array used by process 0 for global velocity I/O */
vect_t *vel = NULL;

void Usage(char* prog_name);
void Get_args(int argc, char* argv[], int* n_p, int* n_steps_p,
    double* delta_t_p, int* output_freq_p, char* g_i_p);
void Get_init_cond(double masses[], vect_t pos[],
    vect_t loc_vel[], int n, int loc_n);
void Gen_init_cond(double masses[], vect_t pos[],
    vect_t loc_vel[], int n, int loc_n);
void Output_state(double time, double masses[], vect_t pos[],
    vect_t loc_vel[], int n, int loc_n);
void Compute_force(int loc_part, double masses[], vect_t loc_forces[],
    vect_t pos[], int n, int loc_n);
void Update_part(int loc_part, double masses[], vect_t loc_forces[],
    vect_t loc_pos[], vect_t loc_vel[], int n, int loc_n, double delta_t);

int main(int argc, char* argv[])
{    int n;    /* Total number of particles */
    int loc_n;    /* Number of my particles */
    int n_steps;    /* Number of timesteps */
    int step;    /* Current step */
    int loc_part;    /* Current local particle */
    int output_freq;    /* Frequency of output */
    double delta_t;    /* Size of timestep */
    double t;    /* Current Time */
    double* masses;    /* All the masses */
    vect_t* loc_pos;    /* Positions of my particles */
    vect_t* pos;    /* Positions of all particles */
    vect_t* loc_vel;    /* Velocities of my particles */
    vect_t* loc_forces;    /* Forces on my particles */
char g_i;        /* G_en or _nput init consds */
double start, finish; /* For timings */

MPI_Init(&argc, &argv);
comm = MPI_COMM_WORLD;
MPI_Comm_size(comm, &comm_sz);
MPI_Comm_rank(comm, &my_rank);

Get_args(argc, argv, &n, &n_steps, &delta_t, &output_freq, &g_i);
loc_n = n/comm_sz; /* n should be evenly divisible by comm_sz */
masses = malloc(n*sizeof(double));
pos = malloc(n*sizeof(vect_t));
loc_forces = malloc(loc_n*sizeof(vect_t));
loc_pos = pos + my_rank*loc_n;
loc_vel = malloc(loc_n*sizeof(vect_t));
if (my_rank == 0) vel = malloc(n*sizeof(vect_t));
MPI_Type_contiguous(DIM, MPI_DOUBLE, &vect_mpi_t);
MPI_Type_commit(&vect_mpi_t);

if (g_i == 'i')
  Get_init_cond(masses, pos, loc_vel, n, loc_n);
else
  Gen_init_cond(masses, pos, loc_vel, n, loc_n);

start = MPI_Wtime();
#endif NO_OUTPUT
Output_state(0.0, masses, pos, loc_vel, n, loc_n);
#endif
for (step = 1; step <= n_steps; step++) {
  t = step*delta_t;
  for (loc_part = 0; loc_part < loc_n; loc_part++)
    Compute_force(loc_part, masses, loc_forces, pos, n, loc_n);
  for (loc_part = 0; loc_part < loc_n; loc_part++)
    Update_part(loc_part, masses, loc_forces, loc_pos, loc_vel, 
    n, loc_n, delta_t);
  MPI_Allgather(MPI_INPLACE, loc_n, vect_mpi_t, 
    pos, loc_n, vect_mpi_t, comm);
#endif NO_OUTPUT
  if (step % output_freq == 0)
    Output_state(t, masses, pos, loc_vel, n, loc_n);
#endif
}

finish = MPI_Wtime();
if (my_rank == 0)
  printf("Elapsed time = %e seconds\n", finish-start);

MPI_Type_free(&vect_mpi_t);
free(masses);
free(pos);
free(loc_forces);
free(loc_vel);
if (my_rank == 0) free(vel);

MPI_Finalize();

return 0;
mpi_nbody_basic.c

} /* main */

/*-----------------------------*/
* Function: Usage
* Purpose: Print instructions for command-line and exit
* In arg:
*   prog_name: the name of the program as typed on the command-line
*/
void Usage(char* prog_name) {
    fprintf(stderr, "usage: mpiexec -n <number of processes> %s\n", prog_name);
    fprintf(stderr, "<number of particles> <number of timesteps>\n");
    fprintf(stderr, "<size of timestep> <output frequency>\n");
    fprintf(stderr, "'g': program should generate init consds\n");
    fprintf(stderr, "'i': program should get init consds from stdin\n");
    exit(0);
} /* Usage */

/*-----------------------------*/
* Function: Get_args
* Purpose: Get command line args
* In args:
*   argc: number of command line args
*   argv: command line args
* Out args:
*   n_p: pointer to n, the number of particles
*   n_steps_p: pointer to n_steps, the number of timesteps
*   delta_t_p: pointer to delta_t, the size of each timestep
*   output_freq_p: pointer to output_freq, which is the number of timesteps between steps whose output is printed
*   g_i_p: pointer to char which is 'g' if the init consds should be generated by the program and 'i' if they should be read from stdin
*/
void Get_args(int argc, char* argv[], int* n_p, int* n_steps_p,
              double* delta_t_p, int* output_freq_p, char* g_i_p) {
    if (my_rank == 0) {
        if (argc != 6) Usage(argv[0]);
        *n_p = strtol(argv[1], NULL, 10);
        *n_steps_p = strtol(argv[2], NULL, 10);
        *delta_t_p = strtod(argv[3], NULL);
        *output_freq_p = strtol(argv[4], NULL, 10);
        *g_i_p = argv[5][0];
    }
    MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
    MPI_Bcast(n_steps_p, 1, MPI_INT, 0, comm);
    MPI_Bcast(delta_t_p, 1, MPI_DOUBLE, 0, comm);
    MPI_Bcast(output_freq_p, 1, MPI_INT, 0, comm);
    MPI_Bcast(g_i_p, 1, MPI_CHAR, 0, comm);
    if (*n_p <= 0 || *n_steps_p < 0 || *delta_t_p <= 0) {
        if (my_rank == 0) Usage(argv[0]);
        MPI_Finalize();
        exit(0);
mpi_nbody_basic.c

    }
    if (*g_i_p != 'g' && *g_i_p != 'i') {
      if (my_rank == 0) Usage(argv[0]);
      MPI_Finalize();
      exit(0);
    }
  # ifdef DEBUG
    if (my_rank == 0) {
      printf("n = %d\n", *n_p);
      printf("n_steps = %d\n", *n_steps_p);
      printf("delta_t = %e\n", *delta_t_p);
      printf("output_freq = %d\n", *output_freq_p);
      printf("g_i = %c\n", *g_i_p);
    }
  # endif
  } /* Get_args */

  /******************************************************************************
  * Function: Get_init_cond
  * Purpose: Read in initial conditions: mass, position and velocity
  * for each particle
  * In args:
  *  n: total number of particles
  *  loc_n: number of particles assigned to this process
  * Out args:
  *  masses: global array of the masses of the particles
  *  pos: global array of positions
  *  loc_vel: local array of velocities assigned to this process.
  *
  * Global var:
  * vel: Scratch. Used by process 0 for global velocities
  */
  void Get_init_cond(double masses[], vect_t pos[],
                      vect_t loc_vel[], int n, int loc_n) {
    int part;

    if (my_rank == 0) {
      printf("For each particle, enter (in order):
    ");
      printf("its mass, its x-coord, its y-coord, ",
    "its x-velocity, its y-velocity\n"");
      for (part = 0; part < n; part++) {
        scanf("%lf", &masses[part]);
        scanf("%lf", &pos[part][X]);
        scanf("%lf", &pos[part][Y]);
        scanf("%lf", &vel[part][X]);
        scanf("%lf", &vel[part][Y]);
      }
    }
    MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
    MPI_Bcast(pos, n, vect_mpi_t, 0, comm);
    MPI_Scatter(vel, loc_n, vect_mpi_t,
                  loc_vel, loc_n, vect_mpi_t, 0, comm);
  } /* Get_init_cond */

  /******************************************************************************
  * Function: Gen_init_cond
  * Purpose: Generate initial conditions: mass, position and velocity
  * For each particle, enter (in order):
  * its mass, its x-coord, its y-coord, its x-velocity, its y-velocity
  * Scratch. Used by process 0 for global velocities
  */
  void Gen_init_cond(double masses[], vect_t pos[],
                      vect_t loc_vel[], int n, int loc_n) {
    int part;

    if (my_rank == 0) {
      if (*g_i_p != 'g' && *g_i_p != 'i') {
        if (my_rank == 0) Usage(argv[0]);
        MPI_Finalize();
        exit(0);
      }
    }
  # ifdef DEBUG
    if (my_rank == 0) {
      printf("n = %d\n", *n_p);
      printf("n_steps = %d\n", *n_steps_p);
      printf("delta_t = %e\n", *delta_t_p);
      printf("output_freq = %d\n", *output_freq_p);
      printf("g_i = %c\n", *g_i_p);
    }
  # endif
  } /* Get_args */
for each particle

In args:
* n: total number of particles
* loc_n: number of particles assigned to this process

Out args:
* masses: global array of the masses of the particles
* pos: global array of positions
* loc_vel: local array of velocities assigned to this process.

Global var:
* vel: Scratch. Used by process 0 for global velocities

Note: The initial conditions place all particles at
* equal intervals on the nonnegative x-axis with
* identical masses, and identical initial speeds
* parallel to the y-axis. However, some of the
* velocities are in the positive y-direction and
* some are negative.

```c
void Gen_init_cond(double masses[], vect_t pos[],
   vect_t loc_vel[], int n, int loc_n) {
   int part;
   double mass = 5.0e24;
   double gap = 1.0e5;
   double speed = 3.0e4;

   if (my_rank == 0) {
      srand(1);
      for (part = 0; part < n; part++) {
         masses[part] = mass;
         pos[part][X] = part*gap;
         pos[part][Y] = 0.0;
         vel[part][X] = 0.0;
         // if (random()/(double) RAND_MAX) >= 0.5)
         if (part & 2 == 0)
            vel[part][Y] = speed;
         else
            vel[part][Y] = -speed;
      }
   }

   MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
   MPI_Bcast(pos, n, vect_mpi_t, 0, comm);
   MPI_Scatter(vel, loc_n, vect_mpi_t,
      loc_vel, loc_n, vect_mpi_t, 0, comm);
} /* Gen_init_cond */
```

Function: Output state
Purpose: Print the current state of the system

In args:
* time: current time
* masses: global array of particle masses
* pos: global array of particle positions
* loc_vel: local array of my particle velocities
* n: total number of particles
* loc_n: number of my particles

/*****
void Output_state(double time, double masses[], vect_t pos[],
    vect_t loc_vel[], int n, int loc_n) {
    int part;

    MPI_Gather(loc_vel, loc_n, vect_mpi_t, vel, loc_n, vect_mpi_t,
        0, comm);
    if (my_rank == 0) {
        printf("%.2fn", time);
        for (part = 0; part < n; part++) {
            //
            printf("%.3f ", masses[part]);
            printf("%.10.3e ", part, pos[part][X]);
            printf("%.10.3e ", pos[part][Y]);
            printf("%.10.3e
", vel[part][X]);
            printf("%.10.3e
", vel[part][Y]);
        }
        printf("n");
    }
} /* Output_state */

/* Function: Compute_force
 * Purpose: Compute the total force on particle loc_part. Don't
 * exploit the symmetry (force on particle i due to
 * particle k) = -(force on particle k due to particle i)
 * In args:
 *   loc_part: the particle (local index) on which we're computing
 *   masses: the total force
 *   pos: global array of particle positions
 *   n: total number of particles
 *   loc_n: number of my particles
 * Out arg:
 *   loc_forces: array of total forces acting on my particles
 * Note: This function uses the force due to gravitation. So
 * the force on particle i due to particle k is given by
 * m_i m_k (s_k - s_i)/|s_k - s_i|^2
 * Here, m_k is the mass of particle k and s_k is its position vector
 * (at time t).
 */
void Compute_force(int loc_part, double masses[], vect_t loc_forces[],
    vect_t pos[], int n, int loc_n) {
    int k, part;
    double mg;
    vect_t f_part_k;
    double len, len_3; fact;

    /* Global index corresponding to loc_part */
    part = my_rank*loc_n + loc_part;
    loc_forces[loc_part][X] = loc_forces[loc_part][Y] = 0.0;
    #ifndef DEBUG
    printf("Proc %d > Current total force on part %d = (%.3e, %.3e)n",
        my_rank, part, loc_forces[loc_part][X],
        loc_forces[loc_part][Y]);
    # endif
for (k = 0; k < n; k++) {
    if (k != part) {
        /* Compute force on part due to k */
        f_part_k[X] = pos[part][X] - pos[k][X];
        f_part_k[Y] = pos[part][Y] - pos[k][Y];
        len = sqrt(f_part_k[X]*f_part_k[X] + f_part_k[Y]*f_part_k[Y]);
        len_3 = len*len*len;
        mg = -G*masses[part]*masses[k];
        fact = mg/len_3;
        f_part_k[X] *= fact;
        f_part_k[Y] *= fact;
        # ifdef DEBUG
        printf("Proc %d > Force on part %d due to part %d = (%.3e, %.3e)\n", my_rank, part, k, f_part_k[X], f_part_k[Y]);
        # endif
        /* Add force in to total forces */
        loc_forces[loc_part][X] += f_part_k[X];
        loc_forces[loc_part][Y] += f_part_k[Y];
    }
} /* Compute_force */

 /***************************************************************************/
 /* Function: Update_part */
 /* Purpose: Update the velocity and position for particle loc_part */
 /* In args: */
 /*  * loc_part: local index of the particle we're updating */
 /*  * masses: global array of particle masses */
 /*  * loc_forces: local array of total forces */
 /*  * n: total number of particles */
 /*  * loc_n: number of particles assigned to this process */
 /*  * delta_t: step size */
 /* */
 /* In/out args: */
 /*  * loc_pos: local array of positions */
 /*  * loc_vel: local array of velocities */
 /* */
 /* Note: This version uses Euler's method to update both the velocity */
 /* and the position. */

 void Update_part(int loc_part, double masses[], vect_t loc_forces[],
                  vect_t loc_pos[], vect_t loc_vel[], int n, int loc_n,
                  double delta_t) {
    int part;
    double fact;

    part = my_rank*loc_n + loc_part;
    fact = delta_t/masses[part];
    # ifdef DEBUG
    printf("Proc %d > Before update of %d: \n", my_rank, part);
    printf(" Position = (%.3e, %.3e)\n", loc_pos[loc_part][X], loc_pos[loc_part][Y]);
    printf(" Velocity = (%.3e, %.3e)\n", loc_vel[loc_part][X], loc_vel[loc_part][Y]);
    printf(" Net force = (%.3e, %.3e)\n", loc_forces[loc_part][X], loc_forces[loc_part][Y]);
    # endif

# endif
loc_pos[loc_part][X] += delta_t * loc_vel[loc_part][X];
loc_pos[loc_part][Y] += delta_t * loc_vel[loc_part][Y];
loc_vel[loc_part][X] += fact * loc_forces[loc_part][X];
loc_vel[loc_part][Y] += fact * loc_forces[loc_part][Y];
#else
printf("Proc %d > Position of %d = (%.3e, %.3e), Velocity = (%.3e,%.3e)\n",
   my_rank, part, loc_pos[loc_part][X], loc_pos[loc_part][Y],
   loc_vel[loc_part][X], loc_vel[loc_part][Y]);
#endif
} /* Update_part */
/* File: mpi_tsp_stat.c */

/* Purpose: Use iterative depth-first search and MPI to solve an instance of the travelling salesman problem. This version partitions the search tree using breadth-first search. Then each process searches its assigned subtree. There is no reassignment of tree nodes. This version also attempts to reuse deallocated tours. The best tour structure is broadcast using a loop of MPI_Bsend. */

/* Compile: mpicc -g -Wall -o mpi_tsp_stat mpi_tsp_stat.c */
/* Usage: mpiexec -n <proc count> mpi_tsp_stat <matrix_file> */

/* Input: From a user-specified file, the number of cities followed by the costs of travelling between the cities organized as a matrix: the cost of travelling from city i to city j is the ij entry. Costs are nonnegative ints. Diagonal entries are 0. */
/* Output: The best tour found by the program and the cost of the tour. */

/* Notes: */
/* 1. Costs and cities are non-negative ints. */
/* 2. Program assumes the cost of travelling from a city to itself is zero, and the cost of travelling from one city to another city is positive. */
/* 3. Note that costs may not be symmetric: the cost of travelling from A to B, may, in general, be different from the cost of travelling from B to A. */
/* 4. Salesperson's home town is 0. */
/* 5. The digraph is stored as an adjacency matrix, which is a one-dimensional array: digraph[i][j] is computed as digraph[1*n + j] */
/* 6. Define STATS at compile time to get some info on broadcasts of best tour costs. */

/* IPP: Section 6.2.11 (pp. 319 and ff.) */
/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

const int INFINITY = 1000000;
const int NO_CITY = -1;
const int FALSE = 0;
const int TRUE = 1;
const int MAX_STRING = 1000;
const int TOUR_TAG = 1;
const int INIT_COST_MSGS = 100;

typedef int city_t;
typedef int cost_t;

typedef struct {
    city_t* cities; /* Cities in partial tour */
    int Count; /* Number of cities in partial tour */
    cost_t cost; /* Cost of partial tour */
}
typedef struct {
    tour_t* list;
    int list_sz;
    int list_alloc;
} stack_struct;
typedef stack_struct* my_stack_t;

/* head refers to the first element in the queue
 * tail refers to the first available slot */
typedef stack_struct* my_queue_t;
#define Queue_elt(queue, i) (queue->list[(queue->head + i) % queue->list_alloc])

/* Global Vars: */
int n; /* Number of cities in the problem */
int my_rank;
int comm_sz;
MPI_Comm comm;
cost_t* digraph;
#define Cost(city1, city2) (digraph[city1*n + city2])
city_t home_town = 0;
tour_t loc_best_tour;
cost_t best_tour_cost;
MPI_Datatype tour_arr_mpi_t; // For storing the list of cities
char* mpi_buffer;

#ifdef STATS
/* For stats */
int best_costs_bcast = 0;
int best_costs_received = 0;
#endif

void Usage(char* prog_name);
void Read_digraph(FILE* digraph_file);
void Print_digraph(void);
void Check_for_error(int local_ok, char message[], MPI_Comm comm);

void Par_tree_search(void);
void Partition_tree(my_stack_t stack);
void Build_init_stack(my_stack_t stack, city_t tour_list[], int my_count);
void Get_global_best_tour(void);
void Create_tour_fr_list(city_t list[], tour_t tour);
void Set_init_tours(int init_tour_count, int counts[], int displacements[],
        int* my_count_p, int** tour_list_p);
void Build_initial_queue(int* queue_list_p, int queue_size,
        int* init_tour_count_p);
void Print_tour(tour_t tour, char* title);
int Best_tour(tour_t tour);
void Update_best_tour(tour_t tour);
void Copy_tour(tour_t tour1, tour_t tour2);
void Add_city(tour_t tour, city_t_t);
void Remove_last_city(tour_t tour);
int Feasible(tour_t tour, city_t_t city);
int Visited(tour_t tour, city_t_t city);
void Init_tour(tour_t tour, cost_t cost);
tour_t Alloc_tour(my_stack_t avail);
void Free_tour(tour_t tour, my_stack_t avail);

void Look_for_best_tours(void);
void Bcast_tour_cost(cost_t tour_cost);
void Cleanup_msg_queue(void);

my_stack_t Init_stack(void);
void Push(my_stack_t stack, tour_t tour); // Push pointer
void Push_copy(my_stack_t stack, tour_t tour, my_stack_t avail);
tour_t Pop(my_stack_t stack);
int Empty_stack(my_stack_t stack);
void Free_stack(my_stack_t stack);
void Print_stack(my_stack_t stack, char title[]);

/* Circular queue */
my_queue_t Init_queue(int size);
tour_t Dequeue(my_queue_t queue);
void Enqueue(my_queue_t queue, tour_t tour);
int Empty_queue(my_queue_t queue);
void Free_queue(my_queue_t queue);
void Print_queue(my_queue_t queue, char title[]);
int Get_upper_bd_queue_sz(void);

long long Fact(int k);

/*--------------------------------------------------------------*/
int main(int argc, char* argv[]) {
    FILE* digraph_file;
    double start, finish;
    int local_ok = 1, one_msg_sz;
    char usage[MAX_STRING];
    char* ret_buf;

    MPI_Init(&argc, &argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &comm_sz);
    MPI_Comm_rank(comm, &my_rank);
    sprintf(usage, "usage: mpiexec -n <procs> %s <digraph file>\n", argv[0]);

    if (my_rank == 0) {
        digraph_file = fopen(argv[1], "r");
        if (digraph_file == NULL) local_ok = 0;
mpi_tsp_stat.c

Check_for_error(local_ok, "Can't open digraph file", comm);
Read_digraph(digraph_file);
if (my_rank == 0) fclose(digraph_file);
#else DEBUG
if (my_rank == 0) Print_digraph();
#endif
loc_best_tour = Alloc_tour(NULL);
Init_tour(loc_best_tour, INFINITY);
#else DEBUG
Print_tour(-1, loc_best_tour, "Local Best tour");
printf("City count = %d\n", City_count(loc_best_tour));
printf("Cost = %d\n\n", Tour_cost(loc_best_tour));
#endif
best_tour_cost = INFINITY;

MPI_Type_contiguous(n+1, MPI_INT, &tour_arr_mpi_t);
MPI_Type_commit(&tour_arr_mpi_t);

MPI_Pack_size(1, MPI_INT, comm, &one_msg_sz);
mpi_buffer =
    malloc(100*comm_sz*(one_msg_sz + MPI_BSEND_OVERHEAD)*sizeof(char));
MPI_Buffer_attach(mpi_buffer,
    100*comm_sz*(one_msg_sz + MPI_BSEND_OVERHEAD));

start = MPI_Wtime();
Par_tree_search();
finish = MPI_Wtime();
Cleanup_msg_queue();
MPI_Barrier(comm);
MPI_Buffer_detach(&ret_buf, &one_msg_sz);

if (my_rank == 0) {
    Print_tour(loc_best_tour, "Best tour");
    printf("Cost = %d\n", loc_best_tour->cost);
    printf("Elapsed time = %e seconds\n", finish-start);
}
#endif STATS
printf("bcasts = %d, costs received = %d\n",
    best_costs_bcast, best_costs_received);
#endif
MPI_Type_free(&tour_arr_mpi_t);
free(loc_best_tour->cities);
free(loc_best_tour);
free(digraph);

MPI_Finalize();
return 0;
} /* main */

/*-----------------------------*/
/* Function:  Init_tour */
/* Purpose:   Initialize the data members of allocated tour */
/* In args:   */
/* cost:     initial cost of tour */
/* Global in:
**mpi_tsp_stat.c**

* n: number of cities in TSP
* Out arg:
* tour
* Local function
*/

```c
void Init_tour(tour_t tour, cost_t cost) {
    int i;

    tour->cities[0] = 0;
    for (i = 1; i <= n; i++) {
        tour->cities[i] = NO_CITY;
    }
    tour->cost = cost;
    tour->count = 1;
} /* Init_tour */
```

/*----------------------------------------
* Function: Read_digraph
* Purpose: Read in the number of cities and the digraph of costs
* In arg: digraph_file
* Globals out:
* n: the number of cities
* digraph: the matrix file
*/

```c
void Read_digraph(FILE* digraph_file) {
    int i, j, local_ok = 1;

    if (my_rank == 0) fscanf(digraph_file, "%d", &n);
    MPI_Bcast(&n, 1, MPI_INT, 0, comm);
    if (n <= 0) local_ok = 0;
    Check_for_error(local_ok, "Number of vertices must be positive", comm);

    digraph = malloc(n*n*sizeof(cost_t));

    if (my_rank == 0) {
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++) {
                fscanf(digraph_file, "%d", &digraph[i*n + j]);
                if (i == j && digraph[i*n + j] != 0) {
                    fprintf(stderr, "Diagonal entries must be zero\n");
                    local_ok = 0;
                }
                else if (i != j && digraph[i*n + j] <= 0) {
                    fprintf(stderr, "Off-diagonal entries must be positive\n");
                    fprintf(stderr, "digraph[%d,%d] = %d\n", i, j, digraph[i*n+j]);
                }
            }
    }
    local_ok = 0;

    Check_for_error(local_ok, "Error in digraph file", comm);
    MPI_Bcast(digraph, n*n, MPI_INT, 0, comm);
} /* Read_digraph */
```

/*----------------------------------------
* Function: Print_digraph
* Purpose: Print the number of cities and the digraph of costs
*/
```c
mpi_tsp_stat.c

*Globals in:
 * n: number of cities
 * digraph: digraph of costs
* Local function
*/

void Print digraph(void) {
    int i, j;

    printf("Order = %d\n", n);
    printf("Matrix = \n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            printf("%2d. ", digraph[i*n+j]);
        printf("\n");
    }
    printf("\n");
} /* Print_digraph */

/*-------------------*/
* Function:       Par_tree_search
* Purpose:        Use multiple threads to search a tree
* In arg:
 * rank: thread rank
* Globals in:
 * n: total number of cities in the problem
* Notes:
 * 1. The Update_best_tour function will modify the global vars
 *  loc_best_tour and best_tour_cost
* /="/n

void Par_tree_search(void) {
    city_t nbr;
    my_stack_t stack; // Stack for searching
    my_stack_t avail; // Stack for unused tours
    tour_t curr_tour;

    avail = Init_stack();
    stack = Init_stack();
    Partition_tree(stack);

    while (!Empty_stack(stack)) {
        curr_tour = Pop(stack);
        # ifdef DEBUG
        Print_tour(curr_tour, "Popped");
        # endif
        if (City_count(curr_tour) == n) {
            if (Best_tour(curr_tour)) {
                # ifdef DEBUG
                Print_tour(curr_tour, "Best tour");
                # endif
                Update_best_tour(curr_tour);
            } else {
                for (nbr = n-1; nbr >= 1; nbr--)
                    if (Feasible(curr_tour, nbr)) {
                        Add_city(curr_tour, nbr);
                        Push_copy(stack, curr_tour, avail);
                        Remove_last_city(curr_tour);
                    }
        } else {
            if (City_count(curr_tour) > n) {
                if (Valid_tour(curr_tour)) {
                    Add_city(curr_tour, n);
                    Push_copy(stack, curr_tour, avail);
                    Remove_last_city(curr_tour);
                } else {
                    Update_best_tour(curr_tour);
                    } else {
                    if (Valid_tour(curr_tour)) {
                        Add_city(curr_tour, n);
                        Push_copy(stack, curr_tour, avail);
                        Remove_last_city(curr_tour);
                    } else {
                        Update_best_tour(curr_tour);
                    }
                }
            } else {
                if (Valid_tour(curr_tour)) {
                    Add_city(curr_tour, n);
                    Push_copy(stack, curr_tour, avail);
                    Remove_last_city(curr_tour);
                } else {
                    Update_best_tour(curr_tour);
                }
            }
        }
    }
}
```
mpi_tsp_stat.c

} 
Free_tour(curr_tour, avail);
)

#endif DEBUG
printf("Proc %d > Done searching\n", my_rank);
#endif
Free_stack(stack);
Free_stack(avail);
MPI_Barrier(comm);
#ifdef DEBUG
printf("Proc %d > Passed barrier\n", my_rank);
#endif
Get_global_best_tour();
#ifdef DEBUG
printf("Proc %d > Returning to main\n", my_rank);
#endif

} /* Par_tree_search */

/***********************************************************************
* Function:  Get_global_best_tour
* Purpose:    Get global best tour to process 0
*/
void Get_global_best_tour(void) {
    struct {
        int cost;
        int rank;
    } loc_data, global_data;
    loc_data.cost = Tour_cost(loc_best_tour);
    loc_data.rank = my_rank;

    /* Both 0 and the owner of the best tour need global_data */
    MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC, comm);
#ifdef DEBUG
    printf("Proc %d > Returned from reduce, rank = %d, cost = %d\n", my_rank,
            global_data.rank, global_data.cost);
#endif
if (global_data.rank == 0) return;
if (my_rank == 0) {
    MPI_Recv(loc_best_tour->cities, n+1, MPI_INT, global_data.rank, 0, comm, MPI_STATUS_IGNORE);
    loc_best_tour->cost = global_data.cost;
    loc_best_tour->count = n+1;
} else if (my_rank == global_data.rank) {
    MPI_Send(loc_best_tour->cities, n+1, MPI_INT, 0, 0, comm);
}
} /* Get_global_best_tour */

***********************************************************************
* Function:  Partition_tree
* Purpose:    Assign each thread its initial collection of subtrees
* In arg:
*    my_rank
* Out args:
*    stack: stack will store each thread's initial tours
* */
void Partition_tree(my_stack_t stack) {
    int my_count, local_ok = 1;
    int queue_size, init_tour_count;
    city_t *queue_list = NULL;
    city_t *tour_list;
    int counts[comm_sz]; /* For scatter */
    int displacements[comm_sz]; /* For scatter */

    if (my_rank == 0) {
        queue_size = Get_upper_bd_queue_sz();
        ifdef DEBUG
            printf("Proc %d > queue_size = %d\n", my_rank, queue_size);
        endif
        if (queue_size == 0) local_ok = 0;
    }
    Check_for_error(local_ok, "Too many processes", comm);

    if (my_rank == 0)
        Build_initial_queue(&queue_list, queue_size, &init_tour_count);
    MPI_Bcast(&init_tour_count, 1, MPI_INT, 0, comm);

    Set_init_tours(init_tour_count, counts, displacements,
                    &my_count, &tour_list);

    MPI_Scatterv(queue_list, counts, displacements, tour_arr_mpi_t,
                 tour_list, my_count, tour_arr_mpi_t, 0, comm);

    Build_init_stack(stack, tour_list, my_count);

    ifdef DEBUG
        Print_stack(stack, "After set up");
    endif
    if (my_rank == 0) free(queue_list);
    free(tour_list);
} /* Partition_tree */

="/-------------------------------------------------------------------------
* Function:    Build_init_stack
* Purpose:    Push the initial tours onto the stack
* In args:
*    tour_list
*    my_count
* Out arg:
*    tour
* Global in:
*    n
*/

void Build_init_stack(my_stack_t stack, city_t tour_list[], int my_count) {
    int i;
    tour_t tour = Alloc_tour(NULL);

    for (i = my_count-1; i >= 0; i-- ) {
        Create_tour_fr_list(tour_list + i*(n+1), tour);
        Push_copy(stack, tour, NULL);
    }
    Free_tour(tour, NULL);
} /* Build_init_stack */
/*-----------------------------*/
* Function: Create_tour_fr_list
* Purpose: Given a list of cities, create a tour struct
* In arg
*   tour_list
* Out arg
*   tour
* Globals in:
*   n
*   digraph
* Note: Assumes tour has been allocated and copies data into it
*/
void Create_tour_fr_list(city_t list[], tour_t tour) {
    int count = 1, cost = 0;
    city_t city1, city2;

    memcpy(tour->cities, list, (n+1)*sizeof(city_t));

    city1 = 0;
    while (count <= n && list[count] != NO_CITY) {
        city2 = list[count];
        count++;
        cost += Cost(city1, city2);
        city1 = city2;
    }
    tour->count = count;
    tour->cost = cost;
} /* Create_tour_fr_list */

/*-----------------------------*/
* Function: Set_init_tours
* Purpose: Determine which tours in the initial queue should be
*          assigned to each process
* In arg:
*   init_tour_count
* Out args:
*   counts
*   displacements
*   my_count_p
*   my_last_tour_p
*   tour_list_p
* Globals in:
*   my_rank
*   comm_sz
* *
* Note: A block partition is used.
*/
void Set_init_tours(int init_tour_count, int counts[], int displacements[],
                     int* my_count_p, city_t** tour_list_p) {
    int quotient, remainder, i;

    quotient = init_tour_count/comm_sz;
    remainder = init_tour_count % comm_sz;
    for (i = 0; i < remainder; i++)
        counts[i] = quotient+1;
    for (i = remainder; i < comm_sz; i++)
        counts[i] = quotient;
*my_count_p = counts[my_rank];
displacements[0] = 0;
for (i = 1; i < comm_sz; i++)
displacements[i] = displacements[i-1] + counts[i-1];

*tour_list_p = malloc((*my_count_p)*(n+1)*sizeof(int));
} /* Set_init_tours */

/**************************************************************************
* Function: Build_initial_queue
* Purpose: Build queue of tours to be divided among processes/threads
* Global Scratch:
*   queue_size
* Out args
*   init_tour_count_p
*   queue_list_p
*
* Note: Only called by one process/thread
*/
void Build_initial_queue(city_t** queue_list_p, int queue_size,
   int* init_tour_count_p) {
   my_queue_t queue;
   int curr_sz = 0, i;
   city_t nbr;
   tour_t tour = Alloc_tour(NULL);
   city_t* queue_list;

   Init_tour(tour, 0);
   queue = Init_queue(2*queue_size);

   /* Breadth-first search */
   Enqueue(queue, tour); // Enqueues a copy
   // printf("Freeing %p\n", tour);
   Free_tour(tour, NULL);
   curr_sz++;
   while (curr_sz < comm_sz) {
      tour = Dequeue(queue);
      // printf("Dequeued %p\n", tour);
      curr_sz--;
      for (nbr = 1; nbr < n; nbr++)
         if (!Visited(tour, nbr)) {
            Add_city(tour, nbr);
            Enqueue(queue, tour);
            curr_sz++;
            Remove_last_city(tour);
         }
      // printf("Freeing %p\n", tour);
      Free_tour(tour, NULL);
   } /* while */

   *init_tour_count_p = curr_sz;

   #ifdef DEBUG
   Print_queue(queue, 0, "Initial queue");
   #endif

   /* Copy the city lists from queue into queue_list */
queue_list = malloc((*init_tour_count_p)*(n+1)*sizeof(int));
    for (i = 0; i < *init_tour_count_p; i++)
        memcpy(queue_list + i*(n+1), Queue_elt(queue,i)->cities,
               (n+1)*sizeof(int));
    *queue_list_p = queue_list;
    Free_queue(queue);
} /* Build_initial_queue */

/*
 * Function:       Best_tour
 * Purpose:        Determine whether addition of the hometown to the
 *                 n-city input tour will lead to a best tour.
 * In arg:         tour: tour visiting all n cities
 * Ret val:        TRUE if best tour, FALSE otherwise
 */
int Best_tour(tour_t tour) {
    cost_t cost_so_far = Tour_cost(tour);
    city_t last_city = Last_city(tour);

    Look_for_best_tours();

    if (cost_so_far + Cost(last_city, home_town) < best_tour_cost)
        return TRUE;
    else
        return FALSE;
} /* Best_tour */

/*
 * Function:       Look_for_best_tours
 * Purpose:        Examine the message queue for tour costs received from
 *                 other processes. If a tour cost that's less than the
 *                 current best cost on this process, best_tour_cost will
 *                 be updated.
 * Global In/out:  best_tour_cost
 * Note:           Tour costs are probed for and received as long as there are
 *                 messages with TOUR_TAG.
 */
void Look_for_best_tours(void) {
    int done = FALSE, msg_avail, tour_cost;
    MPI_Status status;

    while(!done) {
        MPI_Iprobe(MPI_ANY_SOURCE, TOUR_TAG, comm, &msg_avail,
                   &status);
        if (msg_avail) {
            MPI_Recv(&tour_cost, 1, MPI_INT, status.MPI_SOURCE, TOUR_TAG,
                      comm, MPI_STATUS_IGNORE);
            #ifdef STATS
            best_costs_received++;
            #endif
            #ifdef VERBOSE_STATS
            printf("Proc %d > received cost %d\n", my_rank, tour_cost);
            #endif
            if (tour_cost < best_tour_cost) best_tour_cost = tour_cost;
} else {
    done = TRUE;
}
} /* while */
} /* Look_for_best_tours */

#ifdef VERBOSE_STATS
    printf("Proc %d > cost = %d\n", my_rank, best_tour_cost);
#endif
} /* Update_best_tour */

void Bcast_tour_cost(int tour_cost) {
    int dest;

    for (dest = 0; dest < comm_sz; dest++)
        if (dest != my_rank)
            MPI_Bsend(&tour_cost, 1, MPI_INT, dest, TOUR_TAG, comm);
#ifndef STATS
    best_costs_bcast++;
#endif
} /* Bcast_tour_cost */

/* Function:  Copy_tour
* Purpose:   Copy the given tour to the local_best_tour.
* In arg:
*    tour: the given tour.
* Global out:
*    local_best_tour: the current best tour on this process.
* Note:
*  1. If the tour has been received up to this point, local_best_tour will be updated.
*  2. This function is called only if the tour is the best tour. It copies the tour.
*/
void Copy_tour(tour_t tour) {
    Copy_tour(tour, local_best_tour);
}

/* Function:  Update_best_tour
* Purpose:   Update the existing best tour with the input tour +
*            hometown
* In arg:
*    tour: tour that's visited all n-cities
* Global out:
*    local_best_tour: the current best tour on this process
*    best_tour_cost
* Note:
*  1. The input tour hasn't had the hometown added as the last
*     city before the call to Update_loc_best_tour. So we call
*     Add_city(local_best_tour, hometown) before returning.
*  2. This function will only be called if tour has lower cost
*     than any tour local or nonlocal that has been received up
*     to this point. Hence it updates best_tour_cost and broadcasts
*     the best_tour_cost.
*/
void Update_best_tour(tour_t tour) {
    Copy_tour(tour, local_best_tour);
    Add_city(local_best_tour, hometown);
    best_tour_cost = Tour_cost(local_best_tour);
    Bcast_tour_cost(best_tour_cost);
    #ifndef VERBOSE_STATS
    printf("Best tour\n");
    printf("Proc %d > cost = %d\n", my_rank, best_tour_cost);
   #endif
} /* Update_best_tour */
# mpi_tsp_stat.c

* **Purpose:** Copy tour1 into tour2  
* **In arg:**  
  * tour1  
* **Out arg:**  
  * tour2  
*/

```c
void Copy_tour(tour_t tour1, tour_t tour2) {
  // int i;

  memcpy(tour2->cities, tour1->cities, (n+1)*sizeof(city_t));
  // for (i = 0; i <= n; i++)
  //  tour2->cities[i] = tour1->cities[i];
  //tour2->count = tour1->count;
  //tour2->cost = tour1->cost;
} /* Copy_tour */
```

/*****************************/

* **Function:** Add_city  
* **Purpose:** Add city to the end of tour  
* **In arg:**  
  * city  
* **In/out arg:**  
  * tour  
* **Note:** This should only be called if tour->count >= 1.  
*/

```c
void Add_city(tour_t tour, city_t new_city) {
  city_t old_last_city = Last_city(tour);
  tour->cities[tour->count] = new_city;
  (tour->count)++;
  tour->cost += Cost(old_last_city, new_city);
} /* Add_city */
```

/*****************************/

* **Function:** Remove_last_city  
* **Purpose:** Remove last city from end of tour  
* **In/out arg:**  
  * tour  
* **Note:**  
  * Function assumes there are at least two cities on the tour ---  
  * i.e., the hometown in tour->cities[0] won't be removed.  
*/

```c
void Remove_last_city(tour_t tour) {
  city_t old_last_city = Last_city(tour);
  city_t new_last_city;

  tour->cities[tour->count-1] = NO_CITY;
  (tour->count)--;
  new_last_city = Last_city(tour);
  tour->cost -= Cost(new_last_city, old_last_city);
} /* Remove_last_city */
```

/*****************************/

* **Function:** Feasible  
* **Purpose:** Check whether nbr could possibly lead to a better  
* solution if it is added to the current tour. The  
* function checks whether nbr has already been visited  
* in the current tour, and, if not, whether adding the  
* edge from the current city to nbr will result in
int Feasible(tour_t tour, city_t city) {
    city_t last_city = Last_city(tour);

    if (!Visited(tour, city) &&
        Tour_cost(tour) + Cost(last_city, city) < best_tour_cost)
        return TRUE;
    else
        return FALSE;
} /* Feasible */

int Visited(tour_t tour, city_t city) {
    int i;

    for (i = 0; i < City_count(tour); i++)
        if (Tour_city(tour, i) == city) return TRUE;
    return FALSE;
} /* Visited */

void Print_tour(tour_t tour, char* title) {
    int i;
    char string[MAX_STRING];

    if (my_rank >= 0)
        sprintf(string, "Proc %d > %s %p: ", my_rank, title, tour);
    else
        sprintf(string, "%s: ", title);

    for (i = 0; i < City_count(tour); i++)
        sprintf(string + strlen(string), "%d ", Tour_city(tour, i));
    printf("%s\n\n", string);
} /* Print_tour */
MPI_TSPSTAT.C

/*----------------------------------------*/
* Function: Alloc_tour
* Purpose: Allocate memory for a tour and its members
* In/out arg:
*   avail: stack storing unused tours
* Global in: n, number of cities
* Ret val: Pointer to a tour_struct with storage allocated for its
*           members
*/
tour_t Alloc_tour(my_stack_t avail) {
    tour_t tmp;

    if (avail == NULL || Empty_stack(avail)) {
        tmp = malloc(sizeof(tour_struct));
        tmp->cities = malloc((n+1)*sizeof(city_t));
        return tmp;
    } else {
        return Pop(avail);
    }
} /* Alloc_tour */

/*----------------------------------------*/
* Function: Free_tour
* Purpose: Free a tour
* In/out arg:
*   avail
*   tour
*/
void Free_tour(tour_t tour, my_stack_t avail) {
    if (avail == NULL) {
        free(tour->cities);
        free(tour);
    } else {
        Push(avail, tour);
    }
} /* Free_tour */

/*----------------------------------------*/
* Function: Init_stack
* Purpose: Allocate storage for a new stack and initialize members
* Out arg: stack_p
*/
my_stack_t Init_stack(void) {
    int i;

    my_stack_t stack = malloc(sizeof(stack_struct));
    stack->list = malloc(n*n*sizeof(tour_t));
    for (i = 0; i < n*n; i++)
        stack->list[i] = NULL;
    stack->list_sz = 0;
    stack->list_alloc = n*n;

    return stack;
} /* Init_stack */
mpi_tsp_stat.c

* Function: Push
* Purpose: Push a tour pointer onto the stack
* In arg: tour
* In/out arg: stack
*/
void Push(my_stack_t stack, tour_t tour) {
    if (stack->listSz == stack->listAlloc) {
        fprintf(stderr, "Stack overflow in Push!\n");
        free(tour->cities);
        free(tour);
    } else {
        #ifdef DEBUG
        printf("In Push, listSz = %d, pushing %p and %p\n",
            stack->listSz, tour, tour->cities);
        Print_tour(-1, tour, "About to be pushed onto stack");
        printf("\n");
        #endif
        stack->list[stack->listSz] = tour;
        (stack->listSz)++;
    }
} /* Push */

/*-----------------------------------------------------------------------------
* Function: Push_copy
* Purpose: Push a copy of tour onto the top of the stack
* In arg: tour
* In/out arg: stack
* avail
* Error: If the stack is full, print an error and exit
*/
void Push_copy(my_stack_t stack, tour_t tour, my_stack_t avail) {
    tour_t tmp;

    if (stack->listSz == stack->listAlloc) {
        fprintf(stderr, "Stack overflow!\n");
        exit(-1);
    }
    tmp = Alloc_tour(avail);
    Copy_tour(tour, tmp);
    stack->list[stack->listSz] = tmp;
    (stack->listSz)++;
} /* Push_copy */

/*-----------------------------------------------------------------------------
* Function: Pop
* Purpose: Reduce the size of the stack by returning the top
* In arg: stack
* Ret val: The tour on the top of the stack
* Error: If the stack is empty, print a message and exit
*/
tour_t Pop(my_stack_t stack) {
    tour_t tmp;

    if (stack->listSz == 0) {
        fprintf(stderr, "Trying to pop empty stack!\n");
        exit(-1);
mpi_tsp_stat.c

}  

tmp = stack->list[stack->list_sz-1];
stack->list[stack->list_sz-1] = NULL;
(stack->list_sz)--;
return tmp;
} /* Pop */

.isHiddenComment
* Function: Empty_stack
* Purpose: Determine whether the stack is empty
* In arg: stack
* Ret val: TRUE if empty, FALSE otherwise
*/
int Empty_stack(my_stack_t stack) {
    if (stack->list_sz == 0)
        return TRUE;
    else
        return FALSE;
} /* Empty_stack */

.isHiddenComment
* Function: Free_stack
* Purpose: Free a stack and its members
* Out arg: stack
*/
void Free_stack(my_stack_t stack) {
    int i;

    for (i = 0; i < stack->list_sz; i++) {
        free(stack->list[i]->cities);
        free(stack->list[i]);
    }
    free(stack->list);
    free(stack);
} /* Free_stack */

.isHiddenComment
* Function: Print_stack
* Purpose: Print contents of stack for debugging
* In args: all
*/
void Print_stack(my_stack_t stack, char title[])
{
    char string[MAX_STRING];
    int i, j;

    printf("Proc %d > %s\n", my_rank, title);
    for (i = 0; i < stack->list_sz; i++) {
        sprintf(string, "Proc %d > ", my_rank);
        for (j = 0; j < stack->list[i]->count; j++)
            sprintf(string + strlen(string), "%d ", stack->list[i]->cities[j]);
        printf("%s\n", string);
    }
} /* Print_stack */

.isHiddenComment
mpi_tsp_stat.c

* Function: Init_queue
* Purpose: Allocate storage for and initialize data members in
*          new queue
* In arg:  size, the size of the new queue
* Ret val: new queue
*/
my_queue_t Init_queue(int size) {
    my_queue_t new_queue = malloc(sizeof(queue_struct));
    new_queue->list = malloc(size*sizeof(tour_t));
    new_queue->list_alloc = size;
    new_queue->head = new_queue->tail = new_queue->full = 0;

    return new_queue;
} /* Init_queue */

/*---------------------------------------------
* Function: Dequeue
* Purpose: Remove the tour at the head of the queue and return
*          it
* In/out arg: queue
* Ret val: tour at head of queue
*/
tour_t Dequeue(my_queue_t queue) {
    tour_t tmp;

    if (Empty_queue(queue)) {
        fprintf(stderr, "Attempting to dequeue from empty queue\n");
        exit(-1);
    }
    tmp = queue->list[queue->head];
    queue->head = (queue->head + 1) % queue->list_alloc;
    return tmp;
} /* Dequeue */

/*---------------------------------------------
* Function: Enqueue
* Purpose: Add a new tour to the tail of the queue
* In arg:  tour
* In/out arg: queue
*/
void Enqueue(my_queue_t queue, tour_t tour) {
    tour_t tmp;

    if (queue->full == TRUE) {
        fprintf(stderr, "Attempting to enqueue a full queue\n");
        fprintf(stderr, "list_alloc = %d, head = %d, tail = %d\n",
                queue->list_alloc, queue->head, queue->tail);
        exit(-1);
    }
    tmp = Alloc_tour(NULL);
    Copy_tour(tour, tmp);
    // printf("Enqueueing \n", tmp);
    queue->list[queue->tail] = tmp;
    queue->tail = (queue->tail + 1) % queue->list_alloc;
    if (queue->tail == queue->head)
        queue->full = TRUE;
} /* Enqueue */

/* Function: Empty_queue
 * Purpose: Determine whether the queue is empty
 * Ret val: TRUE if queue is empty, FALSE otherwise
 */
int Empty_queue(my_queue_t queue) {
    if (queue->full == TRUE)
        return FALSE;
    else if (queue->head != queue->tail)
        return FALSE;
    else
        return TRUE;
} /* Empty_queue */

/* Function: Free_queue
 * Purpose: Free storage used for queue
 * Out arg: queue
 */
void Free_queue(my_queue_t queue) {
    // int i;
    //
    // for (i = queue->head; i != queue->tail; i = (i+1) % queue->list_alloc) {
    //    free(queue->list[i]->cities);
    //    free(queue->list[i]);
    // }
    free(queue->list);
    free(queue);
} /* Free_queue */

/* Function: Print_queue
 * Purpose: Print contents of queue for debugging
 * In args: all
 */
void Print_queue(my_queue_t queue, char title[]) {
    char string[MAX_STRING];
    int i, j;

    printf("Proc %d > %s\n", my_rank, title);
    for (i = queue->head; i != queue->tail; i = (i+1) % queue->list_alloc) {
        sprintf(string, "Proc %d > %p = ", my_rank, queue->list[i]);
        for (j = 0; j < queue->list[i]->count; j++)
            sprintf(string + strlen(string), "%d ", queue->list[i]->cities[j]);
        printf("%s\n", string);
    }
} /* Print_queue */

/* Function: Get_upper_bd_queue_sz
 * Purpose: Determine the number of tours needed so that
 * each thread/process gets at least one and a level
 * of the tree is fully expanded. Used as upper
 * bound when building initial queue and used as
 * test to see if there are too many threads for
mpi_tsp_stat.c

* the problem size
* Globals In:
*   comm_sz: number of threads
*   n: number of cities
*
*/
int Get_upper_bd_queue_sz(void) {
  int fact = n-1;
  int size = n-1;

  while (size < comm_sz) {
    fact++;
    size *= fact;
  }

  if (size > Fact(n-1)) {
    fprintf(stderr, "You really shouldn't use so many threads for");
    fprintf(stderr, "such a small problem\n");
    size = 0;
  }
  return size;
} /* Get_upper_bd_queue_sz */

/*----------------------------------------------------------
 * Function: Fact
 * Purpose: Compute k!
 * In arg: k
 * Ret val: k!
 */
long long Fact(int k) {
  long long tmp = 1;
  int i;

  for (i = 2; i <= k; i++)
    tmp *= i;
  return tmp;
} /* Fact */

/*----------------------------------------------------------
 * Function: Cleanup_msg_queue
 * Purpose: See what messages are outstanding after termination and
 *         receive them.
 */
void Cleanup_msg_queue(void) {
  int msg_recvd;
  MPI Status status;
  char string1[MAX_STRING];
  int counts[2] = {0, 0};
  char work_buf[100000];

  MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &msg_recvd, &status);
  while (msg_recvd) {
    /* Just receive the message ... */
    MPI_Recv(work_buf, 100000, MPI_BYTE, status.MPI_SOURCE,
      status.MPI_TAG, comm, MPI_STATUS_IGNORE);
    if (status.MPI_TAG == TOUR_TAG)
      counts[1]++;
  }
}
else // Unknown
    counts[0]++;
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &msg_recd, &status);
}
sprintf(string1, "Messages not received: unknown = %d, tour = %d",
    counts[0], counts[1]);
// printf("Proc %d > %s\n", my_rank, string1);
} /* Cleanup_msg_queue */

="/**********************************************************************/
*/ Function: Check_for_error
* Purpose: See if any process has found an error. Terminate
* if there has been an error.
*/
void Check_for_error(
    int    local_ok    /* in */,
    char   message[]   /* in */,
    MPI_Comm comm      /* in */) {
    int ok;

    MPI_Allreduce(&local_ok, &ok, 1, MPI_INT, MPI_MIN, comm);
    if (ok == 0) {
        int my_rank;
        MPI_Comm_rank(comm, &my_rank);
        if (my_rank == 0) {
            fprintf(stderr, "Proc %d > %s\n", my_rank, message);
            fflush(stderr);
        }
        MPI_Finalize();
        exit(-1);
    }
} /* Check_for_error */
1. Consider solving the matrix multiplication problem $A \times B = C$ using MPI. Assume that $A$, $B$, and $C$ are square $n \times n$ matrices, and that we decide to assign each MPI process a block of rows of $C$ to calculate (see diagram). The general steps of the algorithm are:

   **General Steps:**
   I. Root process (rank 0) reads $A$ and $B$ from a file
   II. Distribute $A$ and $B$ matrices to each process
   III. Each process calculates its block of rows of $C$
   IV. Root process (rank 0) collects all parts of $C$
   V. Root process (rank 0) writes $C$ to a file

   C allocated to MPI processes by blocks of rows

   

   a) In Step I why should the 2D matrix $A$ be read from the file and “embedded” into a 1D array in row-major order (i.e., row 0 followed immediately by row 1, followed immediately by row 2, etc.)?

   b) In Step I why should the 2D matrix $B$ be read from the file and “embedded” into a 1D array in column-major order (i.e., column 0 followed immediately by column 1, etc.)?

   c) In Step II what is the best MPI function to use for distributing the necessary parts of $A$ to each MPI process? (Justify why this is the “best” choice of MPI functions)

   d) In Step II what is the best MPI function to use for distributing the necessary parts of $B$ to each MPI process? (Justify why this is the “best” choice of MPI functions)
Unit 4 Quiz Fall 2018

Name:______________________________

c) For Step III write C code using MPI function call(s) that allows each process to determine:
   • the number of MPI processes (e.g., numProc),
   • its own process number/rank (e.g., myID),
   • its startRow of matrix C, and
   • its endRow of matrix C.

Assume argv[1] contains the value of “n”, and that the last process (i.e., numProc-1) receives the remaining rows if n does not divide evenly by numProc.

f) In Step IV what is the best MPI function to use for collecting all parts of C at the Root process (rank 0)? (Justify why this is the “best” choice of MPI functions)