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 \texttt{val} 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?
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 “physics” cluster
- Analyze MPI performance.

To start the lab:
- watch the Lab 10 Video on the eLearning system
- download lab10.zip from the eLearning system and unzipped/extract it locally on your computer
- copy the lab10 directory to briareus.physics.uni.edu using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to briareus.physics.uni.edu using Putty/ssh

Part A: Using an editor on briareus.physics.uni.edu 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 (mpicc -o sum1DArrayA -03 sum1DArrayA.c) and run by: qsub qsub.sum1DArrayA

(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? (change the array size command-line argument in the qsub.sum1DArrayA file and rerun)
**Part B:** Using an editor on briareus.physics.uni.edu open the file `sum1DArraryB.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 (mpicc -o sum1DArraryB -O3 sum1DArraryB.c) and run by: qsub qsub.sum1DArraryB

(4) Why is the parallel sum calculation of `sum1DArraryB.c` faster than `sum1DArraryA.c`?

(4) Change the array size command-line argument in the qsub.sum1DArraryB to 1025 and rerun the program. Why is the parallel sum calculation incorrect?

**Part C:** Copy `sum1DArraryB.c` to `sum1DArraryC.c`. Edit the `sum1DArraryC.c` to distribute the 1D array to MPI processes by MPI_Scatterv. See https://www.cac.cornell.edu/vw/MPIcc/gathersscatter.aspx for information about MPI_Scatterv.

Submit lab10.zip containing question answers and completed program on the eLearning system
Learning Objectives:
- Design an efficient allocation of work to MPI processes in C.
- Write correct C program using MPI library commands to initialize (create) MPI processes, communicate data, and synchronize their operation.
- Time program execution varying the # of MPI processes

Homework #10 Description:
You are to design and write a C program utilizing MPI to efficiently process a sequence Chroma key / Green-screen frames like we did in Homework #8 to create a video of the “Loch Ness monster.” To start the homework, download and extract hw10.zip from the eLearning system, and copy the hw10 directory to briareus.physics.uni.edu using a secure ftp client (winSCP, FileZilla, scp, etc.). The hw10 folder contains:
  - 99 pictures (nessie001.ppm, ..., nessie099.ppm) showing the “Loch Ness monster” moving across a green-screen background (The .ppm format is used here for its simplicity -- [http://www.fileformat.info/format/ppm/egff.htm](http://www.fileformat.info/format/ppm/egff.htm))
  - A picture of “Loch Ness” called swans.ppm that’s to be used as the new background picture replacing the green-screen background in the “nessie” pictures to generate 99 new pictures: frame001.ppm, ..., frame099.ppm
  - A sequential version of the program seqChromaKey.c which takes about 18.5 seconds (on fermil1.cs.uni.edu) to produce < 5 seconds of video (minimum video frames per second, FPS, is about 20 FPS). Clearly too slow for “live” green-screen streaming like you might see in a weather forecast on the local news.
  - My qsub hw10 qsub batch file that you’ll need to customize with your
Your MPI processes can split up the processing of the “nessie” pictures “trivially” based on their “rank” (process id) and the number of picture frames gotten from the command-line argv parameter. I invoked my program in qsub.hw10 as:

```bash
time mpiexec -np ${NUMPROC} /home/fienup/hw10/hw10 /home/fienup/hw10/swans.ppm nessie 99
```

where the command-line parameters are:
- `/home/fienup/hw10/hw10` - complete path name to my executable (you’ll need to change from fienup to your id)
- `/home/fienup/hw10` - complete path name of folder holding the .ppm picture files (again change fienup)
- `swans.ppm` - the name of the new background file
- `nessie` - the base name of the green-screen pictures
- `99` - the number of green-screen pictures

Each of your MPI processes running on nodes in the briareus cluster has access to your NSF-mounted /home/”user” directory. Thus, from the command-line arguments each process can build the file names and open them as needed (e.g., My MPI process with rank 0 opened the file `/home/fienup/hw10/swans.ppm for reading, file /home/fienup/hw10/nessie001.ppm for reading, /home/fienup/hw10/frame001.ppm for writing, etc.).

Complete the following table for your program by varying the qsub.hw10 command: 

<table>
<thead>
<tr>
<th>qsub Command</th>
<th>Number of MPI Processes</th>
<th>Time of sequential seqChromaKey.c (in seconds)</th>
<th>Time of your parallel MPI hw10/chromaKey program</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PBS -l nodes=2:ppn=1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=1:ppn=2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=4:ppn=1</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=2:ppn=2</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=1:ppn=4</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=8:ppn=1</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=4:ppn=2</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Due: Wednesday, Nov. 1 at 5 PM
<table>
<thead>
<tr>
<th>PBS -l nodes=2:ppn=4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBS -l nodes=1:ppn=8</td>
<td>8</td>
</tr>
<tr>
<td>PBS -l nodes=8:ppn=2</td>
<td>16</td>
</tr>
<tr>
<td>PBS -l nodes=4:ppn=4</td>
<td>16</td>
</tr>
<tr>
<td>PBS -l nodes=2:ppn=8</td>
<td>16</td>
</tr>
</tbody>
</table>

Explain your timing results.

Submit hw10.zip containing a completed timing table with explanation and completed program (hw10.c) on the eLearning system.
The MPI cluster we will be using has a "head node" at: briareus.physics.uni.edu

Compiling a MPI program is done using the mpicc 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>PBS Directive</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBS -N mpi</td>
<td>Specifies the job as an MPI job</td>
</tr>
<tr>
<td>PBS -l nodes=4:ppn=2</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.</td>
</tr>
<tr>
<td>PBS -l cput=5:00</td>
<td>Specifies the maximum cpu time the MPI program should be allocated.</td>
</tr>
</tbody>
</table>

### Directions

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir directory greetings:</td>
</tr>
<tr>
<td>cd greetings</td>
</tr>
<tr>
<td>pico greetings.c</td>
</tr>
<tr>
<td>mpicc -o greetings greetings.c</td>
</tr>
<tr>
<td>pico qsub.greetings</td>
</tr>
<tr>
<td>qsub qsub.greetings</td>
</tr>
<tr>
<td>ls</td>
</tr>
<tr>
<td>less mpi.e54321</td>
</tr>
<tr>
<td>less mpi.e54321</td>
</tr>
<tr>
<td>pbsnodes -l free</td>
</tr>
</tbody>
</table>

My `qsub.greetings` file in my `greetings` subdirectory. NOTE: You need to supply the complete path name to the executable MPI program including replacing the `fienup` by your log-on username and specifying the subdirectory containing the executable.

```bash
#!/bin/bash
PBS -N mpi
PBS -l nodes=4:ppn=2
PBS -l cput=5:00
PBS -m be
#
echo "-n"
NUMPROC=`wc -l $(PBS_NODEFILE) | awk '{print $1}'`
#
# Put the full pathname to the executable below
time mpiexec -np $(NUMPROC) /home/fienup/greetings/greetings
```
#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 hostname */
    MPI_Status status; /* return status for receive */

    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);
        }
    }

    MPI_Finalize();
} /* main */
/* File: sum1DArray.c
 * Compile as: mpicc -o sum1DArrayA -O3 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, l, 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 */

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

        printf("length = SIZE\n");
        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);

    //...
/* Send a message with part of array to each MFI 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 %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.**

**Week 10 Monday**

### MPI_Send()

```c
int MPI_Send()
```

- **Arguments**
  - `buffer`: The address of the buffer to send.
  - `count`: The number of data elements to send.
  - `datatype`: The data type of the elements to send.
  - `dest`: The process to which the message is addressed.
  - `tag`: A message tag to distinguish this message from others.

- **Notes**
  - This routine sends a message to another process. It uses blocking semantics, which means that the routine does not return until the message has been sent. It is non-blocking version of the `send` operation.

- **Return value**
  - An MPI error code.

### MPI_Recv()

```c
int MPI_Recv()
```

- **Arguments**
  - `buffer`: The address at which to receive data.
  - `count`: The number of data elements to receive.
  - `datatype`: The data type of the elements to receive.
  - `source`: The process from which the message is received.
  - `tag`: A message tag to distinguish this message from others.

- **Notes**
  - This routine receives data from another process. It uses blocking semantics— it does not return until the message is received. It is a non-blocking version of the `recv` operation, which uses a parameter of type `MPI_Status` that is used to differentiate this receive from other invocations of `MPI_Recv()` when waiting for completion.

- **Return value**
  - An MPI error code.

### MPI_Recv()

```c
int MPI_Finalize()
```

- **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 value**
  - An MPI error code.
Comp. Arch.  

Week 10 Monday

MPI_Scatter could be used to send equal size blocks to each process

```c
int MPI_Scatter(
    void *sendbuffer,      // send buffer
    int *sendcount,        // number of send elements
    MPI_Datatype sendtype, // send type
    int *destbuffer,       // receive buffer
    int *destcount,        // number of receive elements
    MPI_Datatype desttype, // receive type
    int root,              // root process
    MPI_Comm *comm)        // communicator
);
```

Arguments:
- The first three arguments specify the address, size, and type of the data elements to be sent 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_Scatter 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 recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

<table>
<thead>
<tr>
<th>IN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>address of send buffer (choice, significant only at root)</td>
</tr>
<tr>
<td>sendcounts</td>
<td>non-negative integer array (of length group size) specifying the number of elements to send to each processor</td>
</tr>
<tr>
<td>displs</td>
<td>integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i)</td>
</tr>
<tr>
<td>sendtype</td>
<td>data type of send buffer elements (handle)</td>
</tr>
<tr>
<td>recvbuf</td>
<td>address of receive buffer (choice)</td>
</tr>
<tr>
<td>recvcount</td>
<td>number of elements in receive buffer (non-negative integer)</td>
</tr>
<tr>
<td>recvtype</td>
<td>data type of receive buffer elements (handle)</td>
</tr>
<tr>
<td>root</td>
<td>root rank of sending process (integer)</td>
</tr>
<tr>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
</tbody>
</table>
sum1DArrayA.c

#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

    /* 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;
`sum1DArrayA.c`

```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 %.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);
}
} // end main */
```
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 process 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 continue to sends 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 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_Isend()) 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_Bsend()) - 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_Rsend()) 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.

```c
MPI_Group create(MPI_Group old, int size, int rank, MPI_Group *newGroup)

Notes:
Creates a new group by selecting processes from an existing group.
Return value:
An MPI error code.
```

```c
MPI_Comm create(MPI_Comm comm, MPI_Group *group)

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
1. int numCols; /* initialized elsewhere */
2. void broadcast_example()
3. {
4.   int **ranks; /* the ranks that belong to each group */
5.   int myrank;
6.   int rowNumber; /* row number of this process */
7.   int random; /* value that we would like to broadcast */
8.   rowNumber = myRank % numCols;
9.   MPI_Group globalGroup, newGroup;
10.  MPI_Comm comm; (newCom);)
11.  /* initialize ranks[] array */
12.  ranks[0][0] = 0, 1, 2, 3; /* not legal C */
13.  ranks[1][0] = 4, 5, 6, 7;
14.  ranks[2][0] = 8, 9, 10, 11;
15.  ranks[3][0] = 12, 13, 14, 15;
16.  /* Extract the original group handle */
17.  MPI_Comm_create(MPI_COMM_WORLD, &globalGroup);
18.  /* Define the new group */
19.  MPI_Group_newglobal, &numCols, ranks[rowNumber], &newGroup);
20.  /* Create new communicator */
21.  MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);
22.  random = rand();
23.  /* Broadcast 'random' across rows */
24.  MPI_Bcast(&random, 1, MPI_COMM_rowNumber=numCols, newComm)
25. }
```
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>Bitwise and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI_EXOR</td>
<td>Bitwise 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?)


/* File: sum1DArrayB.c
 * Compile as: mpicc -o sum1DArrayB -O3 sum1DArrayB.c
 * Description: An MPI solution to sum a 1D array.
 * Uses group communications:
 * - MPI_Scatterv to send blocks of 1D
 * - MPI_Reduce
 * array to each
 */

#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 for i

    } // end if (myID == RootProcess
// code for all MPI processes.

localArray = (float *) malloc(length_per_process*sizeof(float));
printf("%d
", 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 11 Discussion Questions
Chapter 3.5 – 3.8:

Answer the following textbook questions:

Exercises:

3.16 – hand drawn and scanned is fine

6.15

6.23 – write code, but you don’t need to test it.

3.16. Suppose \texttt{comm.sz} = 8 and the vector \( \mathbf{x} = (0, 1, 2, \ldots, 15) \) has been distributed among the processes using a block distribution. Draw a diagram illustrating the steps in a butterfly implementation of allgather of \( \mathbf{x} \).

6.15. A common problem in MPI programs is converting global array indexes to local array indexes and vice-versa.
   a. Find a formula for determining a global index from a local index if the array has a block distribution.
   b. Find a formula for determining a local index from a global index if the array has a block distribution.
   c. Find a formula for determining a global index from a local index if the array has a cyclic distribution.
   d. Find a formula for determining a local index from a global index if the array has cyclic distribution.

You can assume that the number of processes evenly divides the number of elements in the global array. Your solutions should only use basic arithmetic operators \((+,-,\times,/)\). They shouldn’t use any loops or branches.
6.23. Recall that an MPI_Status object is a struct with members for the source, the tag, and any error code for the associated message. It also stores information on the size of the message. However, this isn’t directly accessible as a member, it is only accessible through the MPI function MPI_Get_count:

```c
int MPI_Get_count(
    MPI_Status* status_p /* in */,
    MPI_Datatype datatype /* in */,
    int* count_p /* out */);
```

When MPI_Get_count is passed the status of a message and a datatype, it returns the number of objects of the given datatype in the message. Thus, MPI_Iprobe and MPI_Get_count can be used to determine the size of an incoming message before the message is received. Use these to write a Cleanup_messages function that can be called before an MPI program quits. The purpose of the function is to receive any unreceived messages so that functions such as MPI_Buffer_detach won’t hang.
Learning Objectives:
- Apply MPI library commands to build a derived datatype that can be used in a collective communications among MPI processes
- Evaluate the performance of MPI_Allgather with block distribution vs. cyclic distribution of an array.

To start the lab:
- watch the Lab 11 Video on the eLearning system
- download lab11.zip from the eLearning system and unzip/extract it locally on your computer
- copy the lab11 directory to briareus.physics.uni.edu using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to briareus.physics.uni.edu using Putty/ssh

Part A: Using an editor on briareus.physics.uni.edu open the file lab11/testSafety.c
This simple MPI program creates two MPI processes: my_rank_of 0 and my_rank_of 1. Both send two messages to the other, then they both receive the message. As discussed in section 3.7.3 “Safety in MPI programs” (pp. 132-134) of the textbook, this program might hang or crash since the MPI standard allows MPI_Send to behave two different way:
- it can copy the message into an MPI-managed buffer and return immediately, or
- it can block until the matching call to MPI_Recv is executed by the receiver.

a) If the later approach is chosen, explain why this program would hang/deadlock.

b) Compile (mpicc -o testSafety testSafety.c) and run by qsub qsub.testSafety
Does this program hang for you?

c) Just because it runs now does not mean that it is safe, the MPI system might switch MPI_Send behavior from buffering to blocking based on system load, message size, etc. To test if a program is safe (i.e., doesn’t rely on MPI-provided buffering), change all the MPI_Send to MPI_Ssend. The MPI_Ssend (synchronous send) always blocks until the matching call to MPI_Recv is executed by the receiver. Edit lab11/testSafety.c to use only MPI_Ssend commands. Compile and re-run the program. Did you see it hang? (Look at mpi.c to see if it ran out of execution time because the program deadlocked).

d) Reorder the MPI_Ssend and MPI_Recv operations to make the program safe. Test that it now finishes as expected.

Part B (Extra Credit): For this lab you are to write the MPI program described in Exercise 6.9 (p. 343) of the textbook. Using an editor on briareus.physics.uni.edu open the file lab11/cyclic_derived.c and using it as your starting point.

Include a report of your findings with timing.

Submit lab11.zip containing question answers and completed program on the eLearning system
6.9. Write an MPI program in which each process generates a large, initialized, \(m\)-dimensional array of doubles. Your program should then repeatedly call `MPI_Allgather` on the \(m\)-dimensional arrays. Compare the performance of the calls to `MPI_Allgather` when the global array (the array that's created by the call to `MPI_Allgather`) has

a. a block distribution, and
b. a cyclic distribution.

To use a cyclic distribution, download the code `cyclic-derived.c` from the book's web site, and use the MPI datatype created by this code for the `destination` in the calls to `MPI_Allgather`. For example, we might call

```c
MPI_Allgather(sendbuf, m, MPI.DOUBLE, recvbuf, l, cyclic_mpi_t, comm);
```

if the new MPI datatype were called `cyclic_mpi_t`.

Which distribution performs better? Why? Don't include the overhead involved in building the derived datatype.
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 hw11.zip from the eLearning system. It contains a “starter” program hw11/hw11.c which you can use as your starting point.

Recall that initially, the 2D-array val 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 and extract the starter code hw11.zip which is available on the eLearning system
2) For this activity I want you to:
   - use FileZilla, WinSCP, ... to copy the starter code hw11 directory (includes nodes file) to bariareus.physics.uni.edu
   - use an editor (emacs or nano) to complete the hw11.c program
   - compile the C to an MPI executable file using: mpicc -o hw11 -03 hw11.c -lm
   - edit the qsub.hw11 file to submit to the queue for execution

My qsub.hw11 file: You’ll need to replace the fienup by your log-on username!
The above runs and times the program on 1024x1024 interior array using 8 MPI processes and threshold of 0.0001 with each MPI process running on a separate computer (node). NOTE: Delete all “old” mpi.o##### and mpi.e##### files except for this run.

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.hw11 command: #PBS -l nodes=4:ppn=2
This means use 4 nodes with 2 processes per node for a total of 8 MPI processes. (This seems to works as expected, except when ppn=1 when the “head-node” seems to run all MPI threads, e.g., #PBS -l nodes=8:ppn=1 runs all 8 MPI threads on the head node)

<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></td>
<td></td>
<td></td>
<td>1024 x 1024</td>
</tr>
<tr>
<td>#PBS -l nodes=2:ppn=1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=1:ppn=2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=4:ppn=1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=2:ppn=2</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=1:ppn=4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=8:ppn=1</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=4:ppn=2</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=2:ppn=4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=1:ppn=8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=8:ppn=2</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=4:ppn=4</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#PBS -l nodes=8:ppn=8</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explain your timing results.

Submit hw11.zip containing a completed timing table with explanation and completed program (hw11.c) on the eLearning system
Lab II Video

\[ P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 \]

Single element output

Composite output

\[ 10 \cdot n = \# \text{ elements per process} \]
cyclic-derived.c

/* File: cyclic-derived.c
 * Purpose: Build a derived datatype that can be used in collective
 * communications of arrays that have a cyclic distribution.
 * The program reads in an array of ints on 0, distributes it
 * among the processes, prints the contents of each process'
 * array, and gathers the array onto 0 and prints it.
 * Compile: mpicc -g -Wall -o cyclic-derived cyclic-derived.c
 * Run: mpiexec -n <number of processes> ./cyclic-derived <n>
 * The global array size, n, should be evenly divisible
 * by the number of processes.
 * Input: n element array of ints
 * Output: The contents of the array on each process
 * The array after it has been gathered on process 0
 * IPP: Exercise 6.9 (p. 343)
 */

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

/* These aren't modified after initialization. */
/* So it's probably safe to make them global. */
int my_rank, comm_sz;
MPI_Comm comm;
MPI_Datatype cyclic_mpi_t;

/* Used as temporary storage on process 0 */
int* scratch = NULL;

void Usage(char* prog_name);
void Get_args(int argc, char* argv[], int* n_p);
void Build_cyclic_mpi_type(int loc_n);
void Get_array(int loc_array[], int n, int loc_n);
void Local_print(int loc_array[], int loc_n);
void Print_loc_arrays(int loc_array[], int n, int loc_n);
void Print_array(int loc_array[], int n, int loc_n);

int main(int argc, char* argv[]) { /* Total number of elements */
    int n;
    int loc_n; /* Number of elements on each process */
    int* loc_array; /* Storage on each process for loc_n ints */

    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);
    loc_n = n/comm_sz; /* n should be evenly divisible by comm_sz */

    loc_array = malloc(loc_n*sizeof(int));
    if (my_rank == 0) scratch = malloc(n*sizeof(int));
cyclic-derived.c

Build_cyclic_mpi_type(loc_n);

Get_array(loc_array, n, loc_n);
// Local_print(loc_array, loc_n);
// if (my_rank == 0) printf("\n");
Print_loc_arrays(loc_array, n, loc_n);
if (my_rank == 0) printf("\n");
Print_array(loc_array, n, loc_n);

MPI_Type_free(&cyclic_mpi_t);
free(loc_array);
if (my_rank == 0) free(scratch);

MPI_Finalize();

return 0;
} /* 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>\n", prog_name);
fprintf(stderr, " n = number of elements in global array\n";
fprintf(stderr, " n should be evenly divisible by comm_sz\n";

} /* Usage */

/************************************************************************
* Function: Get_args
* Purpose: Get command line args
* In args:
*   argc: number of command line args
*   argv: command line args
* Out arg:
*   n_p: pointer to n, the global number of elements
* /
void Get_args(int argc, char* argv[], int* n_p) {

if (my_rank == 0) {
    if (argc != 2) {
        Usage(argv[0]);
        *n_p = 0;
    } else {
        *n_p = strtol(argv[1], NULL, 10);
    }
}
MPI_Bcast(n_p, 1, MPI_INT, 0, comm);

if (*n_p <= 0) {
    MPI_Finalize();
}
cyclic_derived.c

exit(0);
}
} /* Get_args */

/*---------------------------------------------
* Function: Build_cyclic_mpi_type
* Purpose: Build an MPI derived datatype that can be used with
* cyclically distributed data.
* In arg:
* loc_n: The number of elements assigned to each process
* Global out:
* cyclic mpi_t: An MPI datatype that can be used with cyclically
* distributed data
*/
void Build_cyclic_mpi_type(int loc_n) {
    MPI_Datatype temp_mpi_t;
    MPI_Aint lb, extent;

    MPI_Type_vector(loc_n, 1, comm_sz, MPI_INT, &temp_mpi_t);
    MPI_Type_get_extent(MPI_INT, &lb, &extent);
    MPI_Type_create_resized(temp_mpi_t, lb, extent, &cyclic_mpi_t);
    MPI_Type_commit(&cyclic_mpi_t);
} /* Build_cyclic_mpi_type */

/*---------------------------------------------
* Function: Get_array
* Purpose: Read in array of ints on process 0 and distribute
* among processes using a cyclic distribution
* In args:
* n: global number of elements
* loc_n: number of elements sent to each process
* Out arg: loc_array: elements assigned to this process
* Globals in:
* my_rank: process rank
* cyclic mpi_t: derived datatype representing
* data going to each process as it's stored
* on process 0
* Scratch: scratch: used to store array on process 0
* when it's read in
*/
void Get_array(int loc_array[], int n, int loc_n) {
    int i;

    if (my_rank == 0) {
        printf("Enter the %d elements of the array\n", n);
        for (i = 0; i < n; i++)
            scanf("%d", &scratch[i]);
    }
    MPI_Scatter(scratch, 1, cyclic mpi_t, loc_array, loc_n, MPI_INT, 0, comm);
} /* Get_array */

/*---------------------------------------------
* Function: Local_print
* Purpose: Store local array as a string and each process
* prints its own array
* In args:
* loc_array: elements assigned to this process
* loc_n: number of elements sent to each process
*/
cyclic-derived.c

* Globals in:  my_rank: process rank *
*/
void Local_print(int loc_array[], int loc_n) {
  char string[10000];
  int i, offset;

  sprintf(string, "Proc %d > ", my_rank);
  for (i = 0; i < loc_n; i++) {
    offset = strlen(string);
    sprintf(string + offset, "%d ", loc_array[i]);
  }
  printf("%s\n", string);
} /* Local_print */

/*--------------------------------------------------------------------------
* Function:  Print_loc_arrays
* Purpose:  Gather local arrays from all processes onto 0
* and prints them out.
* In args:  loc_array: elements assigned to this process
* n: global number of elements
* loc_n: number of elements sent to each process
* Globals in:  my_rank: process rank
* cyclic_mpi_t: derived datatype representing
data going to each process as it's stored
* on process 0
* Scratch:  scratch: used to store contents of global
* array on process 0
*/
void Print_loc_arrays(int loc_array[], int n, int loc_n) {
  int proc, i;

  MPI_Gather(loc_array, loc_n, MPI_INT, scratch, loc_n, MPI_INT, 0, comm);
  if (my_rank == 0) {
    for (proc = 0; proc < comm_sz; proc++) {
      printf("Proc %d > ", proc);
      for (i = loc_n*proc; i < loc_n*(proc+1); i++)
        printf("%d ", scratch[i]);
      printf("\n");
    }
  } /* Print_loc_arrays */

/*--------------------------------------------------------------------------
* Function:  Print_array
* Purpose:  Print contents of global array.
* In args:  loc_array: elements assigned to this process
* n: global number of elements
* loc_n: number of elements sent to each process
* Globals in:  my_rank: process rank
* cyclic_mpi_t: derived datatype representing
data going to each process as it's stored
* on process 0
* Scratch:  scratch: used to store array on process 0
* when it's read in
*/
void Print_array(int loc_array[], int n, int loc_n) {
  int i;
MPI_Gather(loc.array, loc_n, MPI_INT, scratch, 1, cyclic_mpi_t, 0, comm);
if (my_rank == 0) {
    printf("Array = ");
    for (i = 0; i < n; i++)
        printf("%d ", scratch[i]);
    printf("\n");
}
} /* Print_array */
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:

```
Get input data:
for each timestep {
    if (timestep output) Print positions and velocities of
    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 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:

```
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

from each process.

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.

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_Beast 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?

f) How might we use the tag parameter?
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 from the eLearning system and unzip/extract it locally on your computer
- copy the lab12 directory to briareus.physics.uni.edu using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to briareus.physics.uni.edu using Putty/ssh

Part A: Using an editor on briareus.physics.uni.edu 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 B: 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.

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

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,...,comm.sx -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 comm.sx processes are used?
/* 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 \frac{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(u) is the velocity of the ith particle at time u and */
/* s_i(u) is its position.
#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 _i_nput init conds */
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(vec_t));
loc_forces = malloc(loc_n*sizeof(vec_t));
loc_pos = pos + my_rank*loc_n;
loc_vel = malloc(loc_n*sizeof(vec_t));
if (my_rank == 0) vel = malloc(n*sizeof(vec_t));
MPI_Type_contiguous(DIM, MPI_DOUBLE, &vec mpi_t);
MPI_Type_commit(&vec 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_IN_PLACE, loc_n, vec mpi_t,
    pos, loc_n, vec mpi_t, comm);
  if (my_rank == 0)
    Output_state(t, masses, pos, loc_vel, n, loc_n);
}
finish = MPI_Wtime();
if (my_rank == 0)
  printf("Elapsed time = %e seconds\n", finish-start);

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

MPI_Finalize();

return 0;
/* 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|i>|\n");
    fprintf(stderr, "'g': program should generate initconds\n");
    fprintf(stderr, "'i': program should get initconds 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 conds
 * 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);
if (*g_i_p != 'g' && *g_i_p != 'i') {
    if (my_rank == 0) Usage(argv[0]);
    MPI_Finalize();
    exit(0);
}
#endif 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):\n");
        printf(" its mass, its x-coord, its y-coord, \n");
        printf("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
mpi_nbody_basic.c

* 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.
*/
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) {
    random(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("%.2f\n", time);
        for (part = 0; part < n; part++) {
            //
            printf("%.3f ", masses[part]);
            printf("%.3d %.10.3e ", part, pos[part][X]);
            printf("%10.3e ", pos[part][Y]);
            printf("%.10.3e ", vel[part][X]);
            printf("%10.3e\n", 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
 *          the total force
 * masses: global array of particle masses
 * 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;
    #ifdef 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 DEBUG
        /* 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 DEBUG
    loc_pos[loc_part][X] += loc_vel[loc_part][X]*fact;
    loc_pos[loc_part][Y] += loc_vel[loc_part][Y]*fact;
    loc_vel[loc_part][X] += loc_forces[loc_part][X]*fact;
    loc_vel[loc_part][Y] += loc_forces[loc_part][Y]*fact;
} /* Update_part */
# 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];
#endif DEBUG
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 */
MPI_TSP_STATIC.C

Purpose: Use iterative depth-first search and MPI to solve an
case of the traveling 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.

IPPC: 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 struct {
    tour_t* list;
    int list_alloc;
    int head;
    int tail;
    int full;
} queue_struct;

typedef queue_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;
in 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;

#ifndef 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);
int Visited(tour_t tour, city_t t);
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 && argc != 2) local_ok = 0;
    Check_for_error(local_ok, usage, comm);
    if (my_rank == 0) {
        digraph_file = fopen(argv[1], "r");
        if (digraph_file == NULL) local_ok = 0;
Check for error(local_ok, "Can't open digraph file", comm);
Read_digraph(digraph_file);
if (my_rank == 0) fclose(digraph_file);
#endif DEBUG
if (my_rank == 0) Print_digraph();
#endif

loc_best_tour = Alloc_tour(NULL);
Init_tour(loc_best_tour, INFINITY);
#endif 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_contiguous(&tour_arr_mpi_t);

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

start = MPI_Wtime();
Parallel_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:
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
* */
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])
                }
            }
    }
    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 digraphrix of costs
* In arg:    digraph_file
* */
void Print_digraph(FILE* digraph_file) {
    int i, j;

    if (my_rank == 0) {
        fprintf(digraph_file, "

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
            fprintf(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])
            }
        }
    }
    Check_for_error(local_ok, "Error in digraph file", comm);
    MPI_Bcast(digraph, n*n, MPI_INT, 0, comm);
} /* Print_digraph */
mpiteststat.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
* /
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);
        }
    }
  }
} /* Par_tree_search */
mpi_tsp_stat.c
}

if (curr_tour 
Free_tour(curr_tour, avail);
}

#define DEBUG
printf("Proc %d > Done searching\n", my_rank);
#endif
Free_stack(stack);
Tree_stack(avail);
MPI_Barrier(comm);
#define DEBUG
printf("Proc %d > Passed barrier\n", my_rank);
#endif
Get_global_best_tour();
#define 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_INT, MPI_MINLOC, comm);
#endif 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);
}

/* Function:   Build_init_stack
 * Purpose:    Push the initial tours onto the stack
 * In args:
 *    tour_list
 *    my_count
 * Out arg:
 *    stack
 * 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_frt_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;

    memcopy(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;
}

/*===========================================================================

* 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;
mpi_tsp_stat.c

*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

    /* Freeing %p\n", tour);
    Free_tour(tour, NULL);
    curr_sz++;
    while (curr_sz < comm_sz) {
        tour = Dequeue(queue);
        /* Freeing %p\n", tour);
        curr_sz--;
        for (nbr = 1; nbr < n; nbr++)
            if (!Visisted(tour, nbr)) {
                Add_city(tour, nbr);
                Enqueue(queue, tour);
                curr_sz++;
                Remove_last_city(tour);
            }
        /* 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;
    } /* msg_avail */
    else done = TRUE;
  } /* while */
} /* Look_for_best_tours */
} else {
    done = TRUE;
}
} /* while */
} /* Look_for_best_tours */

/* Function:    Update_best_tour
* Purpose: Replace the existing best tour with the input tour +
*           hometown
* In arg:      tour that's visited all n-cities
* Global out:  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(loc_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, loc_best_tour);
    Add_city(loc_best_tour, hometown);
    best_tour_cost = Tour_cost(loc_best_tour);
    Bcast_tour_cost(best_tour_cost);
    # ifdef VERBOSE_STATS
    Print_tour(loc_best_tour, "Best tour");
    printf("Proc %d > cost = %d\n", my_rank, best_tour_cost);
    # endif
} /* Update_best_tour */

/* Function:    Bcast_tour_cost
* Purpose:     Asynchronous broadcast of tour cost
* Note:        MPI_Bcast is a point of synchronization for the processes.
*               So it can't be used.
*/
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);
    # ifdef STATS
    best_costs_bcast++;
    # endif
} /* Bcast_tour_cost */

/* Function:    Copy_tour
* Purpose:     Copy the current tour to loc_best_tour
* Note:        loc_best_tour is the best tour so far on this process.
*               It is either the input tour, or a copy of the input tour
*               that has been updated by Add_city().
*/
void Copy_tour(tour_t tour, tour_t loc_best_tour) {
    /* Copy the tour to loc_best_tour */
    # ifdef VERBOSE_STATS
    Print_tour(tour, "Local copy");
    # endif
} /* Copy_tour */
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 */

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 */

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)
        snprintf(string, "Proc %d > %s %p: ", my_rank, title, tour);
    else
        snprintf(string, "%s: ", title);

    for (i = 0; i < City_count(tour); i++)
        snprintf(string + strlen(string), "%d ", Tour_city(tour, i));
    printf("%s\n\n", string);
} /* Print_tour */
/* 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
 * Out arg:
 *   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 */
Function: Push
Purpose: Push a tour pointer onto the stack
In arg: tour
In/out arg: stack

```c
void Push(my_stack_t stack, tour_t tour) {
    if (stack->list_sz == stack->list_alloc) {
        fprintf(stderr, "Stack overflow in Push!\n");
        free(tour->cities);
        free(tour);
    } else {
        # ifdef DEBUG
        printf("In Push, list_sz = %d, pushing %p and %p\n",
                stack->list_sz, tour, tour->cities);
        Print tour(-1, tour, "About to be pushed onto stack");
        printf("\n");
        # endif
        stack->list[stack->list_sz] = tour;
        (stack->list_sz)++;
    }
} /* Push */
```

Function: Push_copy
Purpose: Push a copy of tour onto the top of the stack
In arg: tour
In/out arg: stack
* Error: If the stack is full, print an error and exit

```c
void Push_copy(my_stack_t stack, tour_t tour, my_stack_t avail) {
    tour_t tmp;

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

Function: Pop
Purpose: Reduce the size of the stack by returning the top
In arg: stack
* Return: The tour on the top of the stack
* Error: If the stack is empty, print a message and exit

```c
tour_t Pop(my_stack_t stack) {
    tour_t tmp;

    if (stack->list_sz == 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 */

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

******************************************************************************
* Function: Free_stack
* Purpose: Free a stack and its members
* In arg: stack
* 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 */

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

******************************************************************************
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
");
        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
");
        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("Enqueuing %p\n", tmp);
    queue->list[queue->tail] = tmp;
    queue->tail = (queue->tail + 1) & queue->list_alloc;
    if (queue->tail == queue->head)
        queue->full = TRUE;
mpi_tsp_stat.c

} /* 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\n");
        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_recd;
    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_recd, &status);
    while (msg_recd) {
        /* 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);
}
printf(string1, "Messages not received: unknown = %d, tour = %d",
counts[0], counts[1]);
// printf("Proc %d > %s\n", my_rank, string1);
} /* Cleanup_msg_queue */

.getOrElse()
/* 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 2D SOR problem using MPI. We have several options for assigning chunks of the 2D SOR matrix to MPI processes: 2D and 1D block allocations of the matrix to MPI processes.

2D block allocation with MPI processes in a 2D configuration

1D block allocation by rows with MPI processes in a 1D configuration

1D block allocation by columns with MPI processes in a 1D configuration

(a) Why would the 2D block allocation option be a bad choice for the 2D SOR problem?

(b) If the “root MPI process” (myID equal to 0) is responsible for generating the initial 2D SOR matrix (1.0’s down column 0 and 0.0’s everywhere else), why should it embed the 2D SOR matrix in a 1D array using row-major order (i.e., row 0 followed immediately by row 1, followed immediately by row 2, etc.)?

(c) Assume that the “root MPI process” (myID equal to 0) is responsible for generating the initial 2D SOR matrix (1.0’s down column 0 and 0.0’s everywhere else) which it embeds in a 1D array using row-major order. Why is the 1D-block-allocation-by-rows (middle picture above) option better than the 1D-block-allocation-by-columns option?

(d) Which MPI communication function should be used to send chunks of the initial SOR matrix to the MPI processes?
Unit 4 Quiz Fall 2016


<table>
<thead>
<tr>
<th>Step</th>
<th>A high-level algorithm for the 2D SOR problem using MPI is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Generate/distribute chunks of the initial 2D SOR matrix to MPI processes as myVal</td>
</tr>
<tr>
<td>II</td>
<td>Allocate myNew array that's the same size as myVal. Initialize it with 1.0's down column 0 and 0.0's everywhere else</td>
</tr>
<tr>
<td></td>
<td>do</td>
</tr>
<tr>
<td></td>
<td>III Each MPI process “sends” their top row to their “upNeighbor” and their bottom row to their “downNeighbor”</td>
</tr>
<tr>
<td></td>
<td>localDelta = 0.0</td>
</tr>
<tr>
<td></td>
<td>for each row r in myVal chunk do</td>
</tr>
<tr>
<td></td>
<td>for each column c in myVal chunk do</td>
</tr>
<tr>
<td></td>
<td>myNew[r][c] = average of 4 neighboring elements to myVal[r][c]</td>
</tr>
<tr>
<td></td>
<td>if the difference between myVal[r][c] and myNew[r][c] &gt; localDelta then</td>
</tr>
<tr>
<td></td>
<td>localDelta = difference between myVal[r][c] and myNew[r][c]</td>
</tr>
<tr>
<td></td>
<td>end if</td>
</tr>
<tr>
<td></td>
<td>end for</td>
</tr>
<tr>
<td></td>
<td>end for</td>
</tr>
<tr>
<td>IV</td>
<td>Swap pointers to myVal and myNew</td>
</tr>
<tr>
<td>V</td>
<td>Determine the globalDelta to be the maximum of all localDelta values</td>
</tr>
<tr>
<td></td>
<td>while (globalDelta &gt; threshold)</td>
</tr>
<tr>
<td>VI</td>
<td>Combine final chunks of the 2D SOR problem at “root MPI process” (myID equal to 0)</td>
</tr>
</tbody>
</table>

a) For step I, we have two options:
   - have a single MPI process (like the “root MPI process”) be responsible for generating the initial 2D SOR matrix and distributing it to the MPI processes, or
   - have each MPI process be responsible for generating and initializing its own chunk of the 2D SOR matrix.
Which option would be better? (justify your answer)

b) In my solution for step III, I have MPI processes do all of their MPI_Send’s followed by all of their MPI_Recv’s. While this worked on our cluster (briareus), why is this a poor design?

c) For step V, what MPI communication function(s) should be used to determine the globalDelta and redistribute it to all MPI processes?

d) For step VI, what MPI communication function(s) should be used to combine final chunks of the 2D SOR problem at “root MPI process”?