Show Your Work! Point values are in square brackets. There are 25 points possible.

1. In the following expressions a subscript of 2 denotes a binary (base 2) representation and a subscript of 10 denotes a decimal (base 10) representation. For example, 000111_2 = 7_10. All values are unsigned ints. Evaluate each expression. Your answer can be in binary or decimal. You should assume that the computations are taking place on a system that uses 6 bits for an unsigned int.

(a) \(110100_2 \& 101111_2\)
(b) \(101101_2 << 2_{10}\)
(c) \(110100_2 \land 111001_2\)

[3 points]

(a) \(110100_2 \& 101111_2 = 100100_2 = 36_{10}\)
(b) \(101101_2 << 2_{10} = 110100_2 = 52_{10}\)
(c) \(110100_2 \land 111001_2 = 001101_2 = 13_{10}\)

2. Sally has parallelized a serial program. The original serial program has a run-time of 150 seconds, when the input size is \(10^6\). When the input size is \(10^6\), the parallel program has an efficiency of 0.6 with 10 processes.

(a) What is the run-time of the parallel program when it’s run with 5 — should be 10 — processes? Simplify your answer.

(b) Does the program have linear speedup?

[2]

(a) Parallel efficiency is given by the formula

\[ E = \frac{T_{\text{serial}}}{p T_{\text{parallel}}} \]

We are given that \(T_{\text{serial}} = 150\) seconds, \(E = 0.6\), and \(p = 10\). If we solve the equation

\[ 0.6 = \frac{150}{10 T_{\text{parallel}}} \]

for \(T_{\text{parallel}}\), we get \(T_{\text{parallel}} = 25\) seconds.

Note: Because of the mistake in this version of the midterm, everyone received credit for 2a.

(b) The program does not have linear speedup: a program with linear speedup has an efficiency of 1.0, but Sally’s program only has an efficiency of 0.6.
3. Bob has written an MPI program, and he wants to find the run-time of the Zeebl function. So he’s added the following code to the program:

```c
// comm = MPI_COMM_WORLD
double start, finished, my_elapsed, elapsed;
...
start = MPI_Wtime();
// Call to Zeebl
Zeebl(...);
finish = MPI_Wtime();
my_elapsed = finish - start;
MPI_Reduce(&my_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX,
            0, comm);
if (my_rank == 0)
    printf("Zeebl run-time = %.e seconds\n", elapsed);
...
```

After looking at Bob’s code, Sally told Bob that the timing wasn’t quite right: the reported run-time might not be close to the actual parallel run-time.

Sally is right. Modify the code so that it will accurately report the parallel run-time of Zeebl.

[3]

The problem is that the parallel run-time is the elapsed time from the time the first process starts executing to the time the last process finishes executing. Bob’s code just reports the maximum of the elapsed times of the processes.

Bob can fix this problem by adding a call to `MPI_Barrier` immediately before the first call to `MPI_Wtime`. Then all the processes will start at approximately the same time, and the parallel run-time will be the maximum of the elapsed times of the processes.

```c
MPI_Barrier(comm);
start = MPI_Wtime();
// Call to Zeebl
Zeebl(...);
finish = MPI_Wtime();
my_elapsed = finish - start;
MPI_Reduce(&my_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX,
            0, comm);
if (my_rank == 0)
    printf("Zeebl run-time = %.e seconds\n", elapsed);
```
4. The following list has been distributed among four processes and is being sorted using parallel odd-even transposition sort. Show the contents of each process’s sublist after each phase of the sort. [3]

<table>
<thead>
<tr>
<th>Phase</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>2 5 6</td>
<td>4 7 8</td>
<td>1 3 4</td>
<td>2 3 6</td>
</tr>
<tr>
<td>0</td>
<td>2 4 5</td>
<td>6 7 8</td>
<td>1 2 3</td>
<td>3 4 6</td>
</tr>
<tr>
<td>1</td>
<td>2 4 5</td>
<td>1 2 3</td>
<td>6 7 8</td>
<td>3 4 6</td>
</tr>
<tr>
<td>2</td>
<td>1 2 2</td>
<td>3 4 5</td>
<td>3 4 6</td>
<td>6 7 8</td>
</tr>
<tr>
<td>3</td>
<td>1 2 2</td>
<td>3 3 4</td>
<td>4 5 6</td>
<td>6 7 8</td>
</tr>
</tbody>
</table>

In the even phases (0 and 2), processes 0 and 1 are paired and processes 2 and 3 are paired. In the odd phases (1 and 3), processes 0 and 3 are idle, and processes 1 and 2 are paired.

When two processes are paired, they execute a merge-split: each process sends its keys to the paired process. It then merges its list with the received list, keeping the lower or upper half.
5. Find the output of the following MPI program if it’s run with

(a) 1 process [1]
(b) 4 processes [4]

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

int main(int argc, char* argv[]) {
    int p, my_rank, val, temp, dest, src, i;
    MPI_Status status;
    MPI_Comm comm;

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

    val = my_rank + 2;
    printf("Proc %d > val = %d\n", my_rank, val);
    for (i = 1; i < p; i = 2*i) {
        src = (my_rank + p - i) % p;
        dest = (my_rank + i) % p;
        MPI_Send(&val, 1, MPI_INT, dest, 0, comm);
        MPI_Recv(&temp, 1, MPI_INT, src, 0, comm, &status);
        val *= temp;
        printf("Proc %d > temp = %d, val = %d\n", my_rank, temp, val);
    }

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

Continue on the following page.
5. Continued

(a)

\[
\begin{array}{cccccc}
\text{my_rank} & \text{val} & \text{temp} & \text{src} & \text{dest} & i \\
0 & 2 & & & & 1 \\
\end{array}
\]

Output:

Proc 0 > val = 2

Proc 1 > val = 3

Proc 2 > val = 4

Proc 3 > val = 5

Proc 0 > temp = 5, val = 10
Proc 1 > temp = 2, val = 6
Proc 2 > temp = 3, val = 12
Proc 3 > temp = 4, val = 20
Proc 0 > temp = 12, val = 120
Proc 1 > temp = 20, val = 120
Proc 2 > temp = 10, val = 120
Proc 3 > temp = 6, val = 120

Note that the order in which the lines of output from different processes actually appears is random.
6. A Pthreads program contains the following global variables:

```c
int thread_count, total = 0;
char* messages[100];
pthread_mutex_t total_mutex;
```

The main thread gets `thread_count` from the command line and initializes `total_mutex`. It also initializes each entry of `messages` to `NULL`, and then starts the threads. The threads all run the following `Thread_work` function. Find the output of the threads if the main thread starts

(a) 1 thread [1]
(b) 2 threads [2]

```c
void *Thread_work(void* rank) {
    long my_rank = (long) rank;
    int dest = (my_rank + 1) % thread_count;

    messages[dest] = malloc(100*sizeof(char));
    sprintf(messages[dest], "Greetings to %d from %ld", dest, my_rank);

    pthread_mutex_lock(&total_mutex);
    total++;
    pthread_mutex_unlock(&total_mutex);
    while (total < thread_count);

    if (messages[my_rank] != NULL)
        printf("Thread %ld > total = %d, %s\n", my_rank, total,
                messages[my_rank]);
    return NULL;
} /* Thread_work */
```
6. Continued
(a)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Thread 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>0, 1</td>
</tr>
<tr>
<td>messages[0]</td>
<td>NULL, “Greetings…”</td>
</tr>
<tr>
<td>total_mutex</td>
<td>unlocked, locked, unlocked</td>
</tr>
<tr>
<td>dest</td>
<td>0</td>
</tr>
</tbody>
</table>

Output:

Thread 0 > total = 1, Greetings to 0 from 0

(b)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>0, 1, 2</td>
<td></td>
</tr>
<tr>
<td>msgs[0]</td>
<td>NULL, “Greetings…”</td>
<td></td>
</tr>
<tr>
<td>msgs[1]</td>
<td>NULL, “Greetings…”</td>
<td></td>
</tr>
<tr>
<td>ttl_mtx</td>
<td>unl, lck, unl, lck, unl</td>
<td></td>
</tr>
</tbody>
</table>

Output:

Thread 0 > total = 2, Greetings to 0 from 1
Thread 1 > total = 2, Greetings to 1 from 0
7. Write an MPI function `Sum_sq_diff` that takes as input two local arrays, each having `loc_n` elements, and a communicator. The function should compute the differences between corresponding elements of the arrays, square these differences, and form the global sum of the squares of the differences. The function should return this global sum of the squares of the differences. The return value should be the same for all the processes. You can use any of the MPI functions we've discussed in class.

The function prototype is

```c
double Sum_sq_diff(double loc_a[], double loc_b[], int loc_n,
                    MPI_Comm comm);
```

As an example, suppose there are two processes. Also suppose the values of `loc_a` and `loc_b` are

Proc 0: loc_a = {1, 3}, loc_b = {1, 2}
Proc 1: loc_a = {4, 6}, loc_b = {5, 6}

(So `loc_n = 2`.) Then the processes should compute

\[(1 - 1)^2 + (3 - 2)^2 + (4 - 5)^2 + (6 - 6)^2 = 2,\]

and the function should return 2 on each process. [5]

```c
double Sum_sq_diff(double loc_a[], double loc_b[], int loc_n,
                    MPI_Comm comm) {
    int i;
    double diff, my_sum_sq = 0, sum_sq;
    
    for (i = 0; i < loc_n; i++) {
        diff = loc_a[i] - loc_b[i];
        my_sum_sq += diff*diff;
    }
    
    MPI_Allreduce(&my_sum_sq, &sum_sq, 1, MPI_DOUBLE, MPI_SUM, comm);
    return sum_sq;
}
```