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. All values are unsigned 6-bit ints. Evaluate each expression. Your answer should be an unsigned 6-bit binary int. [3 points]

(a) $011010_2 \land 110111_2 = 101101_2$
(b) $111011_2 \gg 2_{10} = 001110_2$
(c) $101001_2 \& 011101_2 = 001001_2$

2. Suppose that each of the arrays $x$ and $y$ stores $n$ doubles. Then the following serial code can be used to implement the vector addition $y = y + x$:

```
for (i = 0; i < n; i++)
    y[i] += x[i];
```

(a) Bob has used MPI to write two parallel versions of this code. The first version uses a block partition of $x$ and $y$. The second uses a cyclic partition. If $n = 10,000,000$ and $p = 4$, which version (if either) would you expect to get better performance? Explain your answer in two sentences or less.

(b) Sally has used Pthreads to write two parallel versions of this code. In both versions $x$, $y$, and $n$ are shared. The first version uses a block partition of $x$ and $y$. The second uses a cyclic partition. If $n = 10,000,000$ and thread_count = 4, which version (if either) would you expect to get better performance? Explain your answer in two sentences or less.

Note that in both parts only the vector addition is being timed. [4]

(a) The two versions should take about the same time. In both the cyclic and the block partitions, the processes will execute something like

```
for (loc_i = 0; loc_i < loc_n; loc_i++)
    loc_y[loc_i] += loc_x[loc_i];
```

Since only the vector addition is being timed, any distribution of the data structures isn’t counted.

(b) The cyclic partition will probably take longer. In a typical system the size of a cache line will be much less than the size of the arrays $x$ and $y$. It will also be greater than 4 doubles. So in the block partition, we expect to get about 1 miss per line for both $x$ and $y$: the thread that “owns” the line will get a miss when it loads the first element of the line. The remaining elements will be hits. For the cyclic partition, however, we expect to get 4 misses per line, since each thread will need to load each line.

In fact, if you run the two versions on a node of the Penguin cluster, the block version takes about 60 milliseconds, and the cyclic version takes about 200 milliseconds.
3. The following MPI function is unsafe (as defined by the MPI Standard). Modify it so that it is safe. [2]

```c
void Average(float vals[], int n, int my_rank, int p, MPI_Comm comm) {
    float temp[MAX]; /* MAX >= n */
    int source, dest, i;

    dest = (my_rank+1) % p;
    source = (my_rank+p-1) % p;

    MPI_Send(vals, n, MPI_FLOAT, dest, 0, comm);
    MPI_Recv(temp, n, MPI_FLOAT, source, 0, comm, MPI_STATUS_IGNORE);
    for (i = 0; i < n; i++)
        vals[i] = (vals[i] + temp[i])/2.0;
}
```

The calls to `MPI_Send` and `MPI_Recv` can be replaced with a single call to `MPI_Sendrecv`:

```c
MPI_Send_recv(vals, n, MPI_FLOAT, dest, 0,
    temp, n, MPI_FLOAT, source, 0, comm, MPI_STATUS_IGNORE);
```

4. An MPI program uses a block distribution of an n-element array. Write a C function `Global_to_local` that converts a global subscript into a local subscript. For example, if n = 12, p = 3, and the global subscript is 7, the function should return 3, since element 7 of the global array will be element 3 in the local array assigned to process 1. You can assume that n is evenly divisible by p. Your solution should only use arithmetic operations. It should not use any MPI functions or any loops. A prototype for the function is

```c
int Global_to_local(int global_subscript, int p, int n);
```

As an example, suppose that n = 12, and p = 3. Then \( loc_n = \frac{12}{3} = 4 \), and the array elements are distributed as follows:

<table>
<thead>
<tr>
<th>Process</th>
<th>Local Subscript</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>1</td>
<td>4 5 6 7</td>
</tr>
<tr>
<td>2</td>
<td>8 9 10 11</td>
</tr>
</tbody>
</table>

So we see that the local subscript is just the remainder of the global subscript divided by \( loc_n \):

```c
int loc_n = n/p;
int local_subscript = global_subscript % loc_n;
return local_subscript;
```
5. Find the output of the following MPI program if it’s run with

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

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

int Assign_val(int my_rank) {
    switch (my_rank) {
        case 0: return 3;
        case 1: return 1;
        case 2: return 4;
        case 3: return 2;
    }
} /* Assign_val */

int main(int argc, char* argv[]) {
    int p, my_rank, val[2], temp[2], dest, 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[0] = Assign_val(my_rank);
    val[1] = my_rank;
    printf("Proc %d > val = %d %d\n", my_rank, val[0], val[1]);
    for (i = 1; i < p; i = 2*i) {
        dest = (my_rank + i) % p;
        MPI_Sendrecv(&val, 2, MPI_INT, dest, 0,
                     &temp, 2, MPI_INT, MPI_ANY_SOURCE,
                     0, comm, &status);
        if (temp[0] < val[0]) {
            val[0] = temp[0];
            val[1] = temp[1];
        }
    }

    printf("Proc %d > val = %d %d\n", my_rank, val[0], val[1]);
    MPI_Finalize();
    return 0;
} /* main */
```

Continue on the following page.
Problem 5 continued.

(a) Trace:

Output:

Proc 0 > val = 3 0
Proc 0 > val = 3 0

(b) Trace:

Output:

Proc 0 > val = 3 0
Proc 1 > val = 1 1
Proc 2 > val = 4 2
Proc 3 > val = 2 3
Proc 0 > val = 1 1
Proc 1 > val = 1 1
Proc 2 > val = 1 1
Proc 3 > val = 1 1
6. A Pthreads program contains the following global variables.

```c
int thread_count, n;
int *list, *new_list, *places;
```

The main thread

1. Initializes the global ints `thread_count` and `n`,
2. Allocates storage for the arrays `list`, `new_list`, and `places`, each of which contains storage for `n` ints,
3. Initializes the elements of `places` to 0,
4. Reads `n` ints into `list`.

After all this setup it starts `thread_count` threads, each of which runs the following thread function. When the thread functions terminate, it prints out `new_list`, free up the storage it allocated and quits.

If `thread_count = 4`, `n = 8`, and `list = {9, 7, 8, 3, 2, 4, 5, 6}`, find the contents of `new_list` after the thread functions terminate. [5]

```c
void *Thread_work(void* rank) {
   long my_rank = (long) rank;
   int local_n = n/thread_count;
   int my_min = my_rank*local_n;
   int my_max = my_min + local_n - 1;
   int i, k;

   for (i = my_min; i <= my_max; i++) {
      // Count number of elements > list[i]
      for (k = 0; k < n; k++)
         if (list[k] > list[i]) places[i]++;
   }
   for (i = my_min; i <= my_max; i++)
      new_list[places[i]] = list[i];

   return NULL;
} /* Thread_work */
```

subscript: 0 1 2 3 4 5 6 7
list: 9 7 8 3 2 4 5 6
places: 0 2 1 6 7 5 4 3
new_list: 9 8 7 6 5 4 3 2
7. Write an MPI function that takes as input arguments a distributed list and a collection of search values, 1 per process. Each MPI process searches its sublist for its value. The function returns a $p$-element `results` array that is identical on each process: if process $q$ found its search value in its sublist, its `results` value is one; otherwise its `results` value is zero.

A function prototype would be

```c
void Search_lists(int sublist[], int sublist_size, int my_value,
                  int results[], MPI_Comm comm);
```

As an example, suppose that $p = 3$, and the following values are input to `Search_lists`:

- Process 0: sublist = {3, 5, 1}, my_value = 2
- Process 1: sublist = {2, 7, 8}, my_value = 7
- Process 2: sublist = {1, 4, 9, 2}, my_value = 2

When `Search_lists` completes, the `results` array should store the three values {0, 1, 1} on each process. Your function should be efficient in its use of communication among the processes. [4]

```c
void Search_lists(int sublist[], int sublist_size, int my_value,
                  int results[], MPI_Comm comm) {
    int i, found = 0;

    for (i = 0; i < sublist_size; i++)
        if (sublist[i] == my_value) {
            found = 1;
            break;
        }

    MPI_Allgather(&found, 1, MPI_INT, results, 1, MPI_INT, comm);
}
```