

Show Your Work! Point values are in square brackets. There are 30 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 ints. Evaluate each expression. Your answer can be in binary or decimal. [3]

(a) $101011_2 \wedge 101101_2 = 000110_2 = 6_{10}$

(b) $111101_2 \ll 2_{10} = 11110100_2 = 244_{10}$. If the computer only stores 6 bits, then the result is $110100_2 = 52_{10}$.

(c) $000110_2 \mid 100111_2 = 100111_2 = 39_{10}$

2. The runtimes (in seconds) of an MPI program are shown in the following table.

p	Input Size n		
	10^6	2×10^6	4×10^6
1	100	200	400
2	60	105	200
4	40	75	120

(a) Find the speedup of the program when $p = 4$ and $n = 4 \times 10^6$.

(b) Find the efficiency of the program when $p = 2$ and $n = 2 \times 10^6$.

Common fractions are fine. [2]

(a)

$$\text{Speedup} = \frac{400}{120} = \frac{10}{3} \approx 3.33$$

(b)

$$\text{Efficiency} = \frac{200}{2 \times 105} = \frac{20}{21} \approx 0.952$$

3. Suppose that each of the arrays `x` and `y` stores `n` doubles. Then the following serial code can be used to implement the dot product of `x` and `y`:

```
dot = 0.0
for (i = 0; i < n; i++)
    dot += x[i]*y[i];
```

Sally has used Pthreads to write two parallel versions of this code. In both versions `x`, `y`, `n`, and `dot` are shared. Each thread computes the dot product of its assigned components storing the result in a private variable. The threads then form the global dot product by adding their private dot products into `dot` in a critical section protected by a mutex. 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 *only* the time spent in the thread function is included in the run-time. [3]

The version that uses the block partition will probably perform better. The reason is that there will probably be more cache misses with the cyclic partition. For example, suppose cache lines are 8 doubles, `x` starts on a cache line boundary, and each thread has its own private cache. Then with a cyclic partition each cache line will be divided among the threads as follows:

Double	0	1	2	3	4	5	6	7
Thread	Th 0	Th 1	Th 2	Th 3	Th 0	Th 1	Th 2	Th 3

So each thread will need to load every cache line. Since there are $10,000,000/8 = 1,250,000$ cache lines in `x`, each thread will need to load 1,250,000 cache lines. On the other hand, with a block partition, each cache line will be loaded by only one thread: the first $1,250,000/4 = 312,500$ will be loaded by thread 0, the next 312,500 by thread 1, etc. So the number of cache line loads with the block partition will be reduced by a factor of four. Similar reasoning applies to `y`.

(On a node of the penguin cluster, the block dot product takes about 37 milliseconds, and the cyclic dot product takes about 140 milliseconds. The cyclic run-time is 3.8 times greater than the block run-time.)

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. [4]

	Process 0	Process 1	Process 2	Process 3
Start	2 4 6	5 7 8	1 3 4	1 2 6
Phase 0	2 4 5	6 7 8	1 1 2	3 4 6
Phase 1	2 4 5	1 1 2	6 7 8	3 4 6
Phase 2	1 1 2	2 4 5	3 4 6	6 7 8
Phase 2	1 1 2	2 3 4	4 5 6	6 7 8

During phases 0 and 2, processes 0 and 1 are paired, and processes 2 and 3 are paired. During phases 1 and 3, processes 0 and 3 are idle, and processes 1 and 2 are paired.

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 p2(a, n) {
    int i;

    for (i = 0; i < n; i++)
        a = 2*a;
    return a;
}

int main(void) {
    int p, my_rank, partner;
    int a, n, t;
    unsigned bitmask = 1;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    n = my_rank + 1;
    a = p2(1, n);
    printf("Proc %d > n = %d, a = %d\n", my_rank, n, a);

    while (bitmask < p) {
        partner = my_rank ^ bitmask;
        MPI_Sendrecv(&n, 1, MPI_INT, partner, 0, &t, 1, MPI_INT,
                    partner, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        a = p2(a, t);
        n += t;
        bitmask <<= 1;
    }

    printf("Proc %d > n = %d, a = %d\n", my_rank, n, a);
    MPI_Finalize();
    return 0;
}
```

(Continue on following page.)

5. (Continued)

(a)

my_rank	p	n	a	bitmask	partner	t
0	1	1	2	1	—	—

Output:

Proc 0 > n = 1, a = 2

Proc 0 > n = 1, a = 2

(b)

my_rank	p	n	a	bitmask	partner	t
0	4	1	2	1 2 4	1 2	2 7
		3	8			
		10	1024			
1	4	2	4	1 2 4	0 3	1 7
		3	8			
		10	1024			
2	4	3	8	1 2 4	3 0	4 3
		7	128			
		10	1024			
3	4	4	16	1 2 4	2 1	3 3
		7	128			
		10	1024			

Output:

Proc 0 > n = 1, a = 2

Proc 1 > n = 2, a = 4

Proc 2 > n = 3, a = 8

Proc 3 > n = 4, a = 16

Proc 0 > n = 10, a = 1024

Proc 1 > n = 10, a = 1024

Proc 2 > n = 10, a = 1024

Proc 3 > n = 10, a = 1024

6. A Pthreads program contains the following global variables.

```
int thread_count;
int* list
sem_t* sems;
```

After getting `thread_count` from the command line, the main thread allocates storage for `thread_count` elements for both `list` and `sems`. It then reads in the contents of `list` and initializes the elements of `sems`. All of the elements of `sems` are initialized to 0, except `sems[0]`, which is initialized to 1. If the following `Thread_work` function is started for each thread by the main thread, find its output if

- (a) `thread_count = 1` and `list = {5}` [1]
- (b) `thread_count = 4` and `list = {5, 4, 3, 2}` [4]

```
void *Thread_work(void* rank) {
    long my_rank = (long) rank;
    int next = my_rank + 1;
    int previous = my_rank - 1;

    sem_wait(&sems[my_rank]);
    if (previous >= 0)
        list[my_rank] += list[previous];
    printf("Thread %ld > After: list[%ld] = %d\n",
        my_rank, my_rank, list[my_rank]);
    if (next < thread_count)
        sem_post(&sems[next]);

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

(a)

thread_count	list	sems	my_rank	next	previous
1	5	1 0	0	1	-1

Output:

Thread 0 > After: list[0] = 5

(Part (b) on next page)

6. continued

thread_count	list	sems	my_rank	next	previous
(b)	5, 4, 3, 2	1, 0, 0, 0	0	1	-1
		0, 0, 0, 0			
		0, 1, 0, 0			
	5, 9, 3, 2	0, 0, 0, 0	1	0	2
		0, 0, 1, 0			
		0, 0, 0, 0			
	5, 9, 12, 2	0, 0, 0, 0	2	1	3
		0, 0, 0, 1			
		0, 0, 0, 0			
	5, 9, 12, 14	0, 0, 0, 0	3	2	4

Output:

```

Thread 0 > After: list[0] = 5
Thread 1 > After: list[1] = 9
Thread 2 > After: list[2] = 12
Thread 3 > After: list[3] = 14

```

7. Write an MPI function that takes as input a distributed list and a set of values, one value 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 finds its search value in its sublist, its `results` value is 1; otherwise its `results` value is zero.

A function prototype would be

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

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.

You can use any MPI function(s) that we've discussed in class. [4]

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

    for (i = 0; i < sublist_size; i++)
        if (sublist[i] == my_value) {
            my_result = 1;
            break;
        }
    MPI_Allgather(&my_result, 1, MPI_INT, results, 1, MPI_INT, comm);
} /* Search_lists */
```

8. A Pthreads program stores a large unsorted array of positive ints that's been initialized by the main thread. Write a thread function that finds the value of the largest element in the array. The list, the number of elements in the list, the number of threads, and the value of the largest element are stored in global variables, which have all been initialized by the main thread. You can also define additional global variables and assume that they're properly initialized by the main thread. Your function should be efficient in dividing the work among the threads.

The header for the function should have the usual prototype:

```
void* Find_max(void* rank);
```

You can assume that the number of threads evenly divides the number of elements in the list. You can also assume that the main thread will print out the maximum value: the thread function doesn't need to print it. Note that you only need to write the thread function: you don't need to write the main function. [4]

```
/* Global vars:
 *
 *   thread_count
 *   n:           Number of elements in list
 *   list
 *   mutex
 *   max:         Since the elements of list are positive, this can
 *                be initialized to 0
 */
void *Find_max(void* rank) {
    long my_rank = (long) rank;
    int local_n = n/thread_count;
    int my_first = my_rank*local_n;
    int my_last = my_first + local_n - 1;
    int i, my_max = 0;

    for (i = my_first; i <= my_last; i++)
        if (list[i] > my_max) my_max = list[i];

    pthread_mutex_lock(&mutex);
    if (my_max > max) max = my_max;
    pthread_mutex_unlock(&mutex);

    return NULL;
} /* Find_max */
```