# CS 484 SP17
# Midterm 2, April 5th, 2017

Total time: 75 mins
No outside materials or devices allowed

Name: ________________________________

NetID: ________________________________

| Question | Total Points | Points Obtained |
|---|---|---|
| Question 1 | 10 | |
| Question 2 | 15 | |
| Question 3 | 15 | |
| Question 4 | 10 | |
| Question 5 | 10 | |
| Question 6 | 15 | |
| Total Points | 75 | |

**1. True or False / Multiple Choice:** Multiple answers may be correct for some questions. (2pts each)

| | |
|---|---|
| a) An algorithm with a larger isoefficiency function (e.g. cubic) is more efficient as the problem size grows than an algorithm with a smaller (say quadratic) isoefficiency function. | True / **False** |
| b) Charm++ divides the work between the system and the programmer such that the programmer can decide how to balance the load, while the runtime system decomposes the computation into parallel pieces. | True / **False** |
| c) By specifying PUP method for your chare class, you are providing the runtime system information that allows it to: | A) Decide the priority of the chare<br>B) Decide which processor to migrate the chare<br>C) **Decide how to copy the object's data into a buffer for migration**<br>D) Decide which method to invoke after load balancing is completed to resume execution |
| d) The efficiency of a parallel program is given by which equation? Assume E = efficiency, S = sequential time, P = parallel time, and N = number of processors. | A) **E = S / (N * P)**<br>B) E = P / (N * S)<br>C) E = S / (N + P)<br>D) E = P / (N + S) |
| e) Which of the following sorting algorithms can be parallelized? | A) Radix Sort B) Histogram Sort<br>C) Quicksort D) **All of the above** |
| f) There are no blocking entry methods in Charm++, i.e., all entry methods are asynchronous and can only have "void" as their return type. | True / False |
| g) Chare method invocations are asynchronous. | **True** / False |
| h) Which MPI call would you use to combine values from all processes and distributes the result back to all processes? | A) **MPI_Allreduce**<br>B) MPI_Broadcast<br>C) MPI_Scatter<br>D) MPI_Alltoall |

**2. Short Answer:**

(5+5+5 points)

Consider the following charm++ program where E is the entry method of a chare. A_Proxy and B_Proxy are proxies to two distinct 1-dimensional chare arrays, The programmer's intent is: send x to the foo method of the 23rd element of the chare array referenced by A_Proxy, and calculate y. E then gets y and sends it to all elements of the chare array referenced by B_Proxy.

```
void E(...) {
    …
    A_Proxy[23].foo(x, &y);
    B_Proxy[ALL].bar(y);
    …
}
```

A. Explain all the errors the programmer has made in the code provided. (Assume all the necessary variables, proxies and entry methods are declared and initialized)

**Errors :**

Methods invocations in charm++ are asynchronous, there is no guarantee that the call to A_proxy will be completed before the calls to B_proxy. The index of A_proxy should be 22 since we are counting from 0. ALL is not a valid keyword for invoking a method on all chares in a chare array.

B. Suggest a solution to fix the issues. You are allowed to change the signatures of *foo*. Full points only if you don't add a new entry method, but correct solutions with additional entry method will still get substantial credit. Describe in pseudocode what *foo* should do.

**Correction :**

One possible solution
Pass B_proxy as a parameter to foo, compute y and use the result as parameter to the bar method of B_proxy

```
A_proxy[22].foo(x,&y,B_proxy b)
{
        y = …..;
        b.bar(y);
}
```

C) Assume a parallel algorithm with problem size $W$ requires $N^3$ operations. The amount of computation on each processor is equal to $\frac{N^3}{P}$. The amount of communication is given by the equation $4 * N$. Compute the isoefficiency of the algorithm.

$$W = N^3$$

$$T_c = \frac{N^3}{P}$$

$$T_p = 4 * N$$

$$T_{t=} T_c + T_p$$

$$T_o = p * T_t - N^3$$
$$= p * 4 * N + N^3 - N^3$$
$$= 4 * p * N$$

$$W = K * T_o$$

$$W = K * 4 * p * N$$

$$N^3 = K * 4 * p * N$$
$$N^2 = K * 4 * p$$

$$= 2\sqrt{Kp}$$

$$W = K * 4 * p * 2 * \sqrt{Kp}$$
$$= 8 * (Kp)^{(\frac{3}{2})}$$

**3. MPI + OpenMP**

(15 pts)

A. Using MPI and OpenMP in conjunction, finish the program below for computing the sum of *f(n)* for each number *n* in a distributed array *globalArray*. For simplicity, assume f() and calculateithValue() are defined elsewhere, and that the total length of the array is exactly divisible by the number of processes.

```
int main() {
  //Initalize MPI stuff
  int world_size, rank, provided;
  MPI_Init_Thread(NULL, NULL, MPI_THREAD_FUNNELED, &provided);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  //Initalize arrays and variables for computation
  const static int array_length = MAX_LENGTH;
  int localLength = array_length/world_size;
  int* localArray = (int*)malloc(sizeof(int)*localLength);
  int *globalArray;
  int globalSum = 0, local_sum = 0;

  //Initialize the array for the first process
  if (rank == 0)
  {
    globalArray = (int*)malloc(array_length*sizeof(int));
    for(int i=0;i<array_length;i++)
      globalArray[i] = calculateithValue(i);
  }

  //Write parallel code that accomplishes the same thing as the
  //  following sequential code. Assume the computations of f() are
     independent :
  int sequentialSum = 0;
  for(int i = 0; i < MAX_LENGTH; ++i)
    sequentialSum += f(globalArray[i]);
```

```
//Write MPI code here to distribute process 0's globalArray to all
other processes' localArrays, then compute partial sums on each
process using OpenMP. Finally, add all processes' local_sum's together
using MPI to compute globalSum. The final globalSum should be
available on process 0.


MPI_Scatter(&global_array[0],localLength,MPI_INTEGER,&localArray[0],lo
calLength,MPI_INTEGER,0,MPI_COMM_WORLD);


int local_sum = 0;

#pragma omp parallel num_threads(nth)
{
      #pragma omp for
      for(int i=0;i<localLength;i++)
      {
            #pragma omp critical
                  local_sum += localArray[i];
      }
}

int global_sum = 0;

MPI_Reduce(&globalSum,&local_sum,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORL
D);

  if (rank == 0)
     assert(globalSum == sequentialSum);

  //Clean up
  free(localArray);
  MPI_Finalize();
  return 0;
}
```

**4. Shared-Memory Atomics**

(10 pts)

Consider the following sequential program. Assume the buffer is bounded, and the size of the buffer is larger than the number of threads. Use c++ atomics for synchronization.

```
int *s = (int *)malloc(MAX_SIZE*sizeof(int));
assert ( s != NULL);

// Keep this loop sequential
for(int i=0;i<MAX_SIZE;i++) s[i] = rand()%1000;

// Re-write this sequential section using threads (pthreads/c++
threads)

int histogram[RANGE];
for(int i=0;i<MAX_SIZE;i++) histogram[s[i]]++;

//Write down your parallel solution here:
//For simplicity,assume MAX_SIZE is exactly divisible by the number of
threads.
int num_threads = atoi(argv[1]);

struct thread_param
{
	int start; int end;
};

int chunk = MAX_SIZE/num_threads;

std::vector<std::thread> workers(num_threads);
std::atomic<int> histogram[RANGE];

for(int i=0;i<RANGE;i++) histogram[i].store(0,memory_order_seq_cst);

for(int i=0;i<num_threads;i++)
{
	thread_param tp;
	tp.start = i*chunk;
	tp.end = std::min((i+1)*chunk,MAX_SIZE);
	std::thread tmp{calc_histogram,tp}
	workers[i] = std::move(tmp);
}
```

```
for(int i=0;i<num_threads;i++) workers[i].join();

void calc_histogram(thread_param tp)
{
	for(int i=tp.start;i<tp.end;i++)
	{
		int pos = s[i];
		histogram[pos].fetch_and_add(1,memory_order_seq_cst);

	}
}
```

**5. MPI_Thread types**

(10 pts)

Describe the difference between MPI_Thread_Single, MPI_Thread_Funneled and MPI_Thread_Multiple. Provide a use case for MPI_Thread_Multiple (the pseudocode or description is sufficient). What is the advantage of using MPI_Thread_Multiple in your use case?

MPI_Thread_Single : One thread executes

MPI_Thread_funneled : Main thread makes MPI calls

MPI_Thread_multiple : multiple threads can make MPI calls

The halo exchange example discussed in lecture is a possible use case.

**6. MPI - Implementation of Broadcast**

(10+5 pts)

Implement a broadcast-like operation on a group of processes. Assume the operation is initiated on the process with rank 0, and the data needs **to reach all other processes.** The broadcast should use a tree algorithm, where the processes are arranged in a binary tree, each process sends data to its children. Children of a process with rank $i$ are the processes with rank $2*i+1$ and $2*i+2$. E.g., process 0 first sends data to 1 and 2, then 1 sends it to 3 and 4, while 2 sends to 5 and 6, and so on.

```
const int ASIZE = ...;
int size, rank;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int bArray[ASIZE];
if(rank == 0) {
  for (int i = 0; i < ASIZE; ++i) {
    bArray[i] = computeithElement(i);
  }
}

// Implement your broadcast algorithm here to send bArray from process
0 to all the other processes in MPI_COMM_WORLD.



int nbr1 = 2*i+1;
int nbr2 = 2*i+2;
int p = (rank-1)/2;

int nbr_count = 0;
if(nbr1 < size)
{
     MPI_Send(&bArray[0],ASIZE,MPI_INTEGER,nbr1,123,MPI_COMM_WORLD);
}
if(nbr2 < size)
{
     MPI_Send(&bArray[0],ASIZE,MPI_INTEGER,nbr2,123,MPI_COMM_WORLD);
}
```

```
if(rank > 0)
{
      MPI_Request r;
      MPI_Status s;
      MPI_Irecv(&bArray[0],ASIZE,MPI_INTEGER,p,123,MPI_COMM_WORLD,r);
      MPI_Wait(r,s);
}


MPI_Finalize();
```

B. What is the communication cost of your implementation? You can assume a communication model where the cost of send/recv is $\alpha + \beta * m$ where $\alpha$ is the latency, $\beta$ is the cost per byte and m is the message size in bytes.

Number of stages = $logP$
Communication cost = $logP * (\alpha + 4 * \beta * ASIZE)$