CS 452 Take Home Final

(Henry, Sam, Maya, N8, Eric, Olivia, Alex, K8, Oumou)

1. [15 points] MPI Coding

Write an MPI function call(s) to do each of the following. You may assume int id, np are the process ID and the total number of processors. Declare local variables as needed.

- **Write code to share 3 variables, int x,y; double z, with all processes. Process 0 has the data.**

```
int x, y;
double z;
MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&y, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&z, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- **Each process has computed an integer vector y of length 10. Combine those vectors so process 0 ends up with both the sum of all those vectors and the total number of zero entries among all the vectors.**

```
vector<int> y(10);
int *reducedY = new int[10];
int numZeros = 0;

for(int i = 0; i < 10; i ++) {
       if(y[i] == 0) {
           numZeros++;
       }
}

MPI_Reduce(numZeros, numZeros, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(y, reducedY, 10, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

- **Create communicators so that each process ends up in exactly one communicator, there are exactly 5 communicators, and the processes are evenly distributed among the communicators as much as possible.**

```
int id;
MPI_Comm_rank(MPI_COMM_WORLD, &id);

int np;
MPI_Comm_size(MPI_COMM_WORLD, &np);

MPI_Comm comm[5];
MPI_Comm_split(MPI_COMM_WORLD,id%5,id,&comm[id%5]);
```

2. **[15 points] Parallel Unordered Search**
   For this question, you will explore algorithms that, when given a list of n items, not in any particular order, and a search key item, will determine if the key item is in the list and if so where.

- **Describe in words a sequential algorithm to find the key, if it exists, in an input list of n items. What is the running time of your algorithm?**
  You would run through a for loop that goes through every element in the input list. Each time you run through, you would check to see if the current element in the list you're checking matches the key. You would keep running through the for loop and only break once you find the element that matches the key, if it exists. The worst-case running time would be O(n) because you may have to run through every element in the list.

- **Describe an algorithm that runs in constant time (O(1)), using O(n) processors, to solve this problem. What type of PRAM are you using?**
  Assign each processor to the index array equal to its id. Each processor will compare the data at that location to the key. There will be an int variable shared between all processors called "found" and initialized to -1. If a processor doesn't find the key at its location, it doesn't change "found". If it does find the value, set "found" to its id, which equals the array index where the item is. If "found" is not -1, the key is at the index equal to "found". This would use CRCW because each individual processor will write to the found only if it finds the key.

- **If you use an EREW PRAM, then the fastest you can solve this problem is O(log n) time. Describe how to do that while using only O(n/log n) processors to keep the total work linear.**
  Make a tree of processors, with log n leaves. Give each of the leaves n/log n array slots to check. They will each check their slots, and if any of them matches the key they will store the index of that slot in a variable. If none of the array elements matches, their variable will equal 0. Then they can send that value up the tree, adding it up with the other processors as they go. Since only one slot can have the key, by the time it reaches the top the value will contain the index of the key. Processor 0 will have to check index 0 to make sure the key isn't there.

  It's using n/log n processors because with n/log n processors you can make a binary tree with log n leaves, and then assign the work evenly among them so that each leaf has n / log n work. So, the bottom layer of the tree can complete in n/log n time, and then in log n time your answers can move up the tree to processor 0.

3. [10 points] Matrix Multiplication

Multiplying two nxn matrices takes O(n^3) time sequentially.

- **(a) Describe the parallel algorithm for matrix multiplication that takes O(n^3) processors and O(log n) time. What kind of PRAM does it use?**

  Using CREW PRAM you can assign each of n^3 processors one cell in each of the two matrices that would intersect during a normal dot product operation (eg: a(i,j) and b(j,k)). Each processor calculates the product of the two cells, the results are summed among all processors with the same value for j and placed into the appropriate square of the resulting matrix.

- **(b) Reduce the number of processors by a factor of log n, to O(n^3/log n), while keeping the O(log n) time bound.**

  The above algorithm can be modified by arranging the processors in a binary tree, with log(n) processors at the bottom of the tree. In this modification, each processor will have to compute n^3/ log n entries in the matrix. Then, you can combine the results of each computation as you go up the binary tree.

4. **[15 points] Maximum Contiguous Subsequence Sum**
   Recall the MCSS problem from CS248 - you are given a list of n integers, which might be both positive and negative, and you are to find a contiguous sublist that maximizes the sum of the items in the sublist. In 248 we wrote three solutions, the last of which ran in O(n) time and used a prefix sum.

- **Describe an algorithm for the MCSS problem using O(log n) time and O(n/log n) processors for the EREW PRAM.**
    - **Hint#1: you can find the sequential code for this problem in Java on the CS248 website.**
    - **https://blue.butler.edu/~jsorenso/oop/code/01/MCSS.java**
    - **Hint#2: a prefix computation can use not only addition for sums, but other operations as well.**

  First calculate a prefix sum in parallel by splitting the array, and then re-combining the values by adding on the values from previous split arrays to the arrays after them, and store the resulting values in an array of size n. Then, construct a binary tree out of the prefix sum values, splitting it in the middle. Check the two halves of the tree, if the maximum subsequence is isolated to one or the other, you know it is from either 0-n/2 or n/2 to n. You can recursively check every subsequence combination going down the tree. If the maximum subsequence falls between the left and right half of the tree, then you can determine it by summing the largest suffix sum on the left, with the largest prefix sum on the right.

5. **[10 points] Parallel Quicksort**

- **If we had 32 processors (and a larger list to sort), how many times would the main loop execute? Suppose we had p processors. Using big-Oh notation, indicate how many main loop iterations there are.**

  With 32 processors
  O((log_2 n)/5)

  With p processors
  O((log_2 n)/ (log_2 p)

- **In the final loop to gather the results, we step through the communicator array backwards. Why do we go backwards and not forwards through this array?**
  - Because MPI_COMM_WORLD is at index 0 and the communicators get more specific with fewer processors in each as it moves through the array. When you try to gather the results, you need to start with the leaf communicators and the move towards the top.

6. **[10 points] Parallel MSTs**
Please answer each of the following questions on our parallel implementation of Sollin's algorithm (Boruvka's Algorithm) for computing the minimum spanning tree of a weighted undirected graph.

- **What is the root[] array used to do?**

  int root[n]; // connected component labels

  Each vertex locates the root of the tree it belongs to. The set of root vertices is the algorithm's input, while the set of non-root vertices is the algorithm's output.

- **What is a star (in the context of this algorithm)? Why do we need to detect stars?**
  The vertice in the center of the minimum spanning tree in which all other vertices are directly connected too is called a star. We need to detect stars because it allows us to find the minimum number of edges by the unique stars.

- **The main loop runs until there is only one connected component to the MST. If the graph begins with n vertices, and hence n components, how many times, in the worst case, does the main loop run? Use big-Oh notation for your answer.**
  Sequential running time O((logV) * E) where V is equal to the number of vertices and E is the number of edges. Sollin's algorithm reduces the “forest” to at most half of the options for each connected component.

Sequential running time:

$$\Theta\left((\log \underset{\substack{\uparrow\\ \text{\# of vertices}}}{V}) \cdot \underset{\substack{\uparrow\\ \text{\# of edges}}}{E}\right)$$