Clone Wars Full of Goodness

Programming Project 3 (major)

due Wednesday, 27 March 2019 by class time

When last you narrated, you had resolved to become a leaf in the river of the Big ~~Dumb~~ Quest and float your way to fame and fortune on the backs of those ~~sniveling~~ chivalrous heroes. You just wanted to merge your way to a speaking part in the sure-to-be *major* motion picture that would be on every *project*or in Middle Earth. On an important side note, it bugs you to consider why people say Middle Earth. Earth is neither the median or mean planet: that honor belongs to Mars, Jupiter, or some terrifying mutant love child of the two.

You have discovered that being nice is really inconvenient and not at all rewarding, because you don't value people that are not you. For example, you helped throw away a few dead orc bodies from the last engagement (with a punctured spleen mind you!) and all you got was a "My most gracious thanks, small creature." WTF is that? Where are the riches, the fame, the season tickets for the Atlanta Falcons? (Oh wait, those are probably not worth anything, so nevermind.) Those orcs smell!

Eventually, you realized that being good may in fact be a way to achieve your goals, but the rewards come far too slowly for you. After all, you have lots of ideas about how to abuse your ultimate power once you get it, and you're worried you may not have time to implement it all if it takes forever to get there.

So you found yourself a tome of Forbidden Ancient Spells of Totally Excellent Results that you thought you could leverage to do things so much FASTER. The basic plan was to clone yourself like in Star Wars, and then have each of your clones going around and getting goodness and giving it to you, the overseer. It's a brilliant plan, so you immediately put it into action.

Job Details. Your task is to write a C++ program that implements clone wars, by writing a "parallel" version of the *mergesort* algorithm. For this algorithm, you must write your code using the following function calls:

```
void mergesort (int * a, int first, int last);
void smerge(int * a, int first1, int last1, int first2, int last2);
int rank(int * a, int first, int last, int valToFind);
void pmerge(int * a, int first, int last, int mid);
```

The parameter `a` will be an array defined dynamically in the main program based on a user-inputted size. You'll need to fill that array with random numbers using the techniques from the first project. I would recommend making sure that the numbers in the array are fairly small, say between 0 and 100. The `first`, `last`, and `mid` variables indicate the range of the arrray you're working on. For the `rank` function, the `valToFind` variable keeps track of which number you want to compute the rank for, and the return value is the number of items that are *less than* the value `valToFind`. (Remember that the array `a` should not have any duplicate entries, so that your code is a bit simpler.)

The parallel mergesort algorithm we learned in class assumes that you will be using shared memory. Since we are using MPI, we will have to "fake" shared memory to do this project. I

recommend that you have process 0 create the initial array `a` and broadcast it to every other process. That way, everyone starts off with the same unsorted array.

To finish the project successfully, you will need to implement the `smerge` function (already done from the prior project, once you modify the parameters to fit the new form) and the `rank` function. The `rank` function can be written independently from its application, in other words, all one needs as input is a sorted array `a` and a value that you want to search for, and it should give the right output.

One trick you might benefit from is the idea of creating an alias for *part* of an array. For example, let's look at the following code:

```
int input[100];

int * a = &input[0];
int * b = &input[50];
```

Now, `a` and `b` will respectively refer to the first and last half of array `input`. You can use this trick to simplify which arrays and which parts of arrays you are looking at. It may help you. Don't forget to thoroughly test your `smerge` and `rank` functions before you move on, because if they don't work, you will be very sad later.

Once you've implemented the functions above, it's time to do the real project, which is to implement `pmerge`. When you write this function, I recommend that you use a testing environment that will limit your errors to one recursion level. This way, you won't have nightmares debugging multiple levels of recursion at once. Here's the setup:

1. When you create your initial array on process 0, make it size 64. Go ahead and sort the first half and the second half. So your input array will be two sorted arrays of size 32 that are glued together. In fact, you might even want to make sure to use the same array each time so that you can work out the exact solution for you example completely. This will be helpful in the debugging step. (To have a fixed array each time, just set a specific seed of your choice and then print out the array to see what you got. You'll get the same array each time with the same seed!)

2. Temporarily, write a base case in your mergesort function that says that if the size is ≤ 32, do nothing. You can of course assume this, because you know that the left and right sides are already sorted.

3. Now, as you're writing your `pmerge` function, you can test it on just the top level, and not have to worry about any recursion calls.

4. I would recommend starting with 4 processors. Just stick with that number for now.

When you go to write the `pmerge` function, you will need to write the many stages of the algorithm so that it works as intended. You will need to make sure to write some code, TEST IT THOROUGHLY, and write the next bit of code. And the TEST THAT THOROUGHLY. If you're not testing, tracing, and working through your code, you will have big problems and you will be sad. Here are the phases of the algorithm that you need to write:

1. First, calculate `SRANKA` and `SRANKB`. You will need to calculate this in parallel by striping the work of computing ranks among all the processors. Remember that you won't be calculating all the ranks (that would be `RANKA` and `RANKB`), but you still need to stripe the samples among all the processors.

2. Share all the `SRANKA` and `SRANKB`, so that everyone has the same answers.

3. Make sure the answers from above are right for your example. TEST THOROUGHLY.

4. Put the sampled rank elements in the correct place an array that will have the correct solution. Let's call this array `WIN`. You know which positions to put the sampled elements in because of the reduction from ranking to merging. TEST THOROUGHLY.

5. Play the shape game. For this, you will have to figure out how to determine a shape according to the algorithm. This is hard to work through, but hugely rewarding when you figure it out. Then, code up dividing up the processing of the shapes so that it can be done in parallel. What processing needs to be done on each shape? That's right, we're doing some `smergeing`. TEST THOROUGLY.

6. Have each process put their smerged shapes in the right place, and then share all of them among everyone. There is an easy trick that lets everyone share their array using `MPI_Reduce`. Ask me about it if you can't figure it out, once you're to this point. Hey, also, test thoroughly.

Your project, in total, is worth 60 project points, which is double of the previous projects. I strongly recommend working in groups of 2 on this project, because you will need someone else who can be sad with you when you are sad. This may be the last time that you ever experience the company of another human, because soon, you will graduate, and then you will have no more friends.

I can also serve as a human who can help you in times of great sadness. I'm around to help with questions.