Homework 1 Report

Jacob Hedén Malm
jacmalm@kth.se
980405-1499

The first homework problem I completed was the matrix sum problem. In order to solve a, I simply added 6 variables, 2 to store the value of the minimum and maximum element, and 2 each to store their x y position. In the for loop that iterates over the matrix and does the actual work, I added two if statements that checked if the current value was less/greater than the value which was currently stored in the min/max values. If it was, I updated the two values.

To solve constraint b, I made these variables global. I then added a function that updates these values when it is called. Each thread computes its maximum and minimum value, as well as the total of its strip, and calls the function which updates the global variables as needed. I used a mutex to make sure that the global variables were not updated by more than one thread at a time.

To solve constraint c, I added a function called bagoftasks, and a global counter. When a thread calls bag of tasks, it returns an integer which is the number of the row that the thread is to compute the values of. After each consecutive call, the global counter increments. If the global counter is not less than the variable size, the thread exits, as the whole matrix has been computed.

The second homework problem I computed was quicksort. I took inspiration from a sequential quicksort algorithm, and started by writing the partition algorithm. This algorithm receives two indexes of an array, looks at the value of the number at the largest index of the array, and uses this as a pivot. It then iterates over the array, comparing each number to the pivot value. It swaps the numbers it iterates over in a way such that at the end of the iteration, the pivot value holds the position it will have in the final sorted array.

Sequential quicksort is a recursive divide and conquer algorithm: it repeats the above mentioned algorithm for the smaller and smaller subarrays to the left and to the right of the pivot variable.

I parallelized quicksort by spawning one thread after each iteration of partition. After quicksort calls partition, we have two partitions of the array that we need to recursively sort. Sequentially, there are two recursive calls. However, each time the algorithm would have recursed twice normally, I recursed once, and spawned another thread to sort the other subarray.
I unfortunately did not time the execution of the sequential versions of the program, so I did not have any values to compare my benchmarks with. I will make sure to do this next time. However, my quicksort algorithm sorted 100 numbers in 0.012 seconds.