Jensen Eicher
Parallel Computing
2/6/20

HW1 Report

After looking over the starter code for homework 1 and reviewing in class material I started by **decomposing** the program into pieces. The first was the linear portion where I wrote two separate "for loops." One to create an array of size 1000 and populate it with random numbers ranging from 0-999. Next, I wrote the linear version of both requested tasks (find max and find sum). For this task I used a single for loop containing and if-statement for finding the max and a simple addition of the current element with all previous to collect the sum. The portion I chose to parallelize was the for loop that found the max and sum as well as the initialization of the array. The next step was beginning implementation with **orchestration**. My initial thought process was to create an array with the same number of elements as threads I used for both the sum and the max. Then, at the end I would combine/compare the remaining elements into the final answer. After attending office hours, I was told that I could simply use private and shared variables to achieve this same concept but allow openMP to separate the variables on its own. This allowed my code to **assign** each thread with an equal portion of the array we were looping over. The **mapping** of the particular threads to the hardware based on the #pragma statements was done by the compiler at run time. The final iteration of my code was of the following architecture.

1. Create an array of N elements with randomly generated values between 0 and 999.
2. Loop through the array and find the max and sum of the elements sequentially (serial method).
3. Spawn 4 threads and create two shared variables (trueSum and trueMax) along with two private variables (threadSum and threadMax).
4. Perform an atomic update to add each thread's individual threadSum to the trueSum one by one (atomic update insures only one thread is touching the memory locations of threadSum and trueSum at once).
5. Use a critical block to force only one thread at a time to compare each of their threadMaxes. (this ensures that a stale value isn't read by a thread).
6. Within the critical block atomically write the threadMax to trueMax if threadMax is greater.
7. Before exiting the parallel portion add all individual threadSums to accumulate the total trueSum.
8. Print and display both sums, maxes and execution times.