Homework 2

Ludvig Larsson

1 Task 1 - Parallel Matrix Computations

1.1 Implementation

Most of the code for this task was already given. The task for me was to find a way to share two variables holding the maximum and minimum value and its corresponding indices in the 2 dimensional array (matrix). I did this by implementing a struct holding value, row and column and instantiating them. The synchronization used could be implemented with `reduction(...)` but I chose to do it by using critical sections. The technique I use, to allow for better overall performance, is to check the current max or min value without any locks. In case it evaluates to true I enter a critical section and evaluate the check again to make sure no data race occurs. This prevents threads from taking the lock when their current value is already smaller for example, despite the possibility that another thread is currently updating it. Though that will not matter since the data race will only allow for even bigger values. Therefore I avoid taking the critical section and increase performance.

1.2 Performance

The performance is quite good. I'm running the program on a quad-core machine and for rather small matrices the speedup is nowhere to be seen. The matrix is too small and sequential execution is faster than creating threads and giving for example 10 elements to each thread or something. When the matrix is 1000x1000 I achieve 2.5x speedup when using 4 threads. For some reason I cannot really explain the speedup continues for additional threads used. It might be that it gives better performance when outer loop is split up additionally rather than already existing threads iterating. I'm not sure. I think the division of iteration in the for loop construct might affect this. If one thread has to take the lock often to update the max or min struct it will affect execution time in comparison to many more threads that will in worse case have to use the lock less often. The overall execution time depends after all on the thread with highest workload considering the static scheduling I believe, each thread gets an equal amount of iterations to complete regardless of time consumption.

2 Task 2 - Parallel Quicksort

Quicksort was quite easy to parallelize. The main takeaway is to let the partial arrays be executed by a task team. The initial call to quicksort from main is `single` but is wrapped in an `omp parallel` which uses as many threads as set by `set_num_threads(...)` `beforehand`. When the pivot element is computed and the two index intervals are determined, each recursive call with a sub-array is a task handled by a thread available in the task group.

2.1 Performance

Performance is quite good. An array of 10 million elements are sorted in 3 seconds and it can be parallelized to achieve speedup of factor 3, that is limited by my quad-core machine at 4-6 threads. In this case too, I don't know why this is occuring. I still believe iterations can be more time consuming than creating tasks.