Pthreads
Programming of Parallel Computers

R.G.A. Deckers

1. *Notes on Divide-and-conquer*

Divide and conquer seems to perform best, as can be seen in the performance section. It seemed to benefit greatly for more threads, capping out at a speedup of about 4, higher performance of about 5 has been achieved when using a ridiculous amount of threads. This is probably due to the fact that, especially later in the lagorithm, the division of work between threads becomes unequal. Meaning a lot of threads will be doing nothing. Low size performance was greatly improved by specifying a (sub-)array size for which we allow forking. Tuning this parameter turned out to be difficult, it is not as small as one would expect from test runs with a maximum fork depth of 1 (i.e. max 2 threads). The current number was found empirically to be the smallest number which didn't affect the perfomance at high array sizes.

I also noticed that there was a much greater spread in runtime for this implementation then there was for the others.

2. *Notes on task based Divide-and-conquer*

An extra I included for fun, I had started working on a threadpool implementation based upon some of my coursework for this course and advanced computer Architecture (as well as the book "The Art of Computer Programming"). However, it does not perform very well yet. I will continue to research why this is the case. (the code seems to be bugged in some way, it works on my Intel laptop but core-dumps on the school's AMD machines. An interesting bug.)

I decided to still include it as a proof-of-concept.

3. *Notes on Peer based algorith*

The peer based algorithm gives the best performance with 8 threads (== hardware thread count), unlike the Divide-and-conquer method each thread seems to work about equally. The overall performance is still a bit lower than the Divide-and-conquer method. This is probably caused by a non-ideal division of work between the threads, however no speedup was obtained by adding more threads. I expect that ideally we should combine this algorith with some kind of work-stealing tasks, giving us the automatic balancing of work-stealing tasks and the even division of work from the peering (meaning we need to execute less steals, and therefore get better performance), but I have been unable to test this yet.

As for how the binning is done, because we are sorting an array of uniform numbers we simply divide the range [0,1] into N_{threads} equal sized bins. We could probably get better and portable performance if we made some sort of histogram first, and then try collecting neighbouring bins (by means of a greedy algorithm for example) until we have a roughly equal distribution between all bins. This would add additional overhead, both in terms of the work required for scanning the array and synchronizing between threads to negotiate binning.

In our implementation each thread writes back it's own results to the original array. We prevent synchronization issues by spinning if we reach our write-back point before all threads have read the original array and copied their elements.

4. *Results*

In figure 1 I have plotted the runtime of all our implementations, in 2 the speedup. Tests were run on my own laptop, a quad-core hyperthreaded i7. Speedup was achieved in going from 4 to 8 threads, this is most likely because the algorithm seems to be limited by cache dramatically, leaving enough room for queueing multiple reads and instructions with the help of hyperthreading.

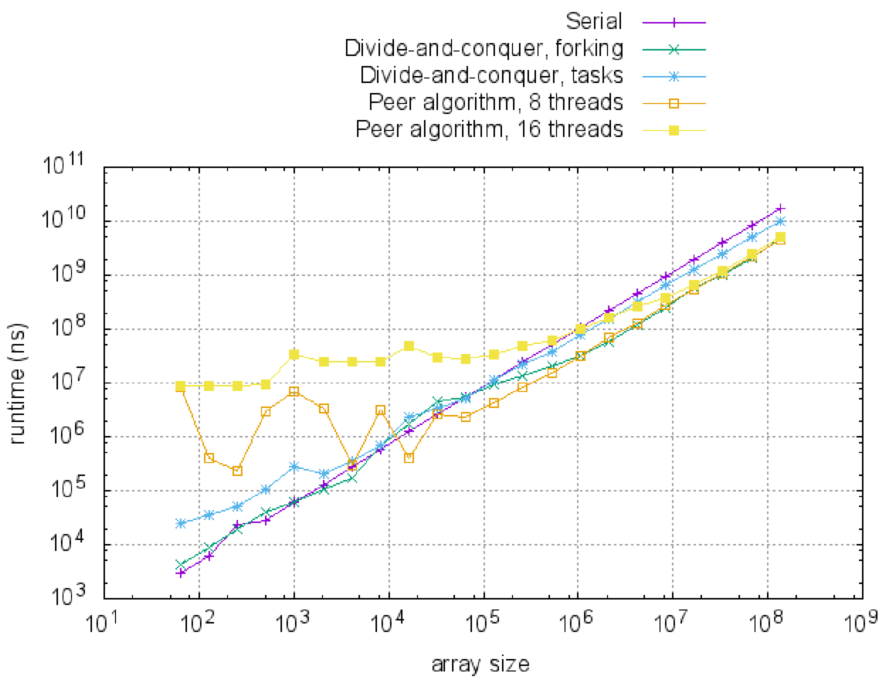

Figure 1: Log-log of the timings of our programs for various array sizes.

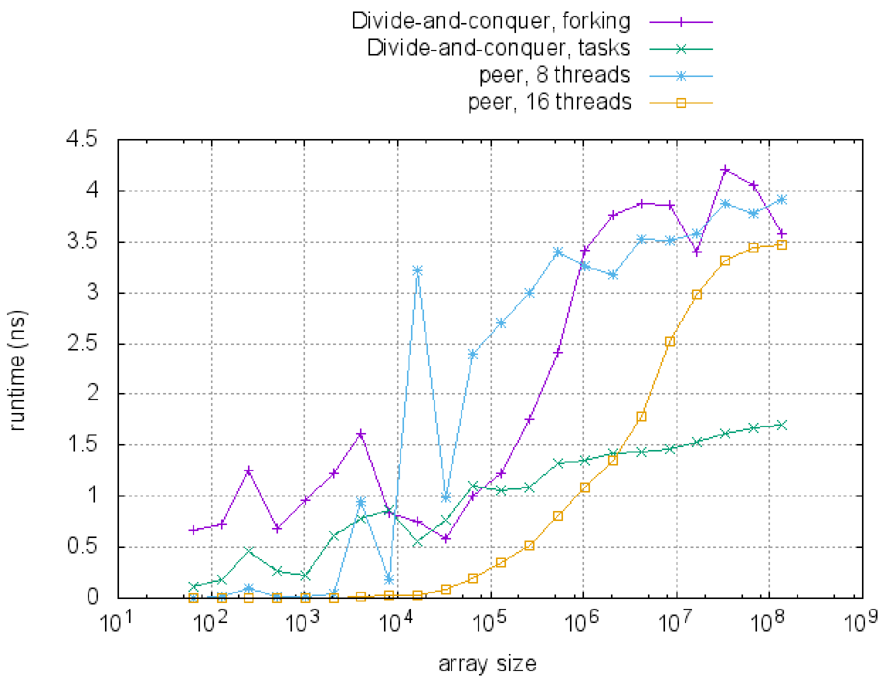

Figure 2: Log-log of the speedup of our programs for various array sizes.

5. *Conclusion*

At the moment, the Divide-and-conquer method (where we spawn a lot of threads), works best. However, I still have faith that this implementation could be beaten by making more efficient use of the different threads by using a hybrid-model (do whatever leads to the best ratio of saturation vs. little-synchronization/context-switching) based on tasks. :)