Parallelized Quicksort with OpenMP and PThreads

Brett Regnier

Introduction

This project is based around the paper *A User's Experience with Parallel Sorting and OpenMP* by Michael Süß and Claudia Leopold, published by the University of Kassel. In their paper they cover how Quicksort can be implemented using OpenMP as well as different speed up techniques of the algorithm based around parallelism. They also showed the difference between OpenMP and PThreads. Their results show that parallel Quicksort can achieve quite the speed up as it can be nearly fully parallelized. The different techniques they implemented between the versions didn't seem to have much improvement, even one version caused an increased amount of time. The authors also made a version that bypasses the OpenMP framework and instead manually construct Quicksort with PThreads. One huge difference I will be making between my code and their code is I want to focus more on the speed up before using an optimization flag on the compiler so I can get an accurate measure if my changes have made any difference In strategies I will be addressing all of their functions by implementing them and attempting to improve the speed.

The functions in my code are similarly named to those in the paper but they have differences including version number updates. Renames are as follows: "sort_seq_1.5" → "SerialQuickSort1_5", "sort_seq_1.5" → "SerialQuickSort2_6", "sort_omp_1.0" → "StackQuickSort3_0", "sort_omp_2.0" → "StackQuickSort4_0", "sort_omp_nested_1.0" → "NestedOMPSort2_0", "sort_omp_taskq_1.0" → "TaskQueueSort2_0", "sort_pthreads_1.0" → "PThreadsSort_2.0", "sort_pthreads_cv_1.0" → "PThreadsSort", "sort_pthreads_yield_1.0" → "PThreadsSort". When discussing my code, I will refer to them as such.

Strategies

A big difference I want to distinguish is using a compiler without the optimization flag, particularly because the compiler could create a lot of improvements behind the scenes that aren't directly caused due to the changes in the code. It is already difficult to improve the speed of Serial Recursive Quicksort, as it is an inherently cache friendly because it utilizes an in-place array and is continuously accessing it every iteration. Therefore, I noticed in their code that std::swap(...) could be replaced with using an inline swap. This would asymptotically improve the speed because it would reduce the amount of branching required by the std::swap function.

This is, I predict, the improvement from “sort_seq_1.5” → “sort_seq_1.6”. Another change I made, which increased the speed of all of the algorithms functions, was replacing the use of a vector with an array. An array, as a primitive type, is faster than a declared type like vector. The speedup is small, more of an asymptotic speed up than a big-O saving. However, without using the “-O3” optimization flag the rate for the “sort_seq_1.6” the computation speed nearly halved with the changes I made in “SerialQuickSort2_6”. However, after recompiling with the “-O3” flag there was almost no difference in speed. For the stack quick sort (“sort_omp_1.0”), I found that replacing the std::pair with properly placed pushes to the stack with primitive types were more effective than using a declared type, without compiler optimization “-O3”. Giving a small asymptotic speed up. Overall, it appears that the only speeds up that can be gained are small ones after the “-O3” flag is applied as the compiler is smart enough to replace most parts that slow down the execution. Therefore, no matter the changes I make only small gains will be made as it is difficult to make Big-O changes. Lastly, I decided on the threshold when the switch to insert by executing a loop statement that finds the best time at which switching is the fastest, which I found at 39 remaining elements.

Experiments

Performance was carried out all on a mostly idle intel i7-4790K CPU, Table 5 & Table 6,with 4 cores and 4 logical cores, giving a total of 8 threads at 4.00GHz baseline, although clock speed ups can occur automagically by Intel’s Turbo Boost technology.The experiments were run on the compiled code with both the “-O3” flag set, and with no optimization flag set during compilation. The goal of this is to show the difference in the computation time from code based on their examples the optimization computation time made afterwards. As well, I averaged the time of my experiments to find a steady medium, while the paper runs the experiments at least 3 times and chooses the best time achieved.

Table 1 and 2, and graph 1 and 2 show the wall-clock time for the experiments run on the unoptimized version with and without the “-O3” flag during compilation. Following the same experiment parameters as the paper, each experiment will run 3 times, each sorting 100 million random integers per iteration. All of the threaded versions did better the more threads were implied, however, “Stack QuickSort” versions 1, and 2 don’t have much improvement from 4 threads to 8 threads. This occurs pretty similarly with “PThreads Quicksort v1.0”.

Table 3, and 4, and graph 3, and 4 show the wall-clock time for the experiments run on versions that I optimized. As mentioned above, I didn’t try to optimize “SerialQuickSort1_5” and therefore the values will be exactly the same. In these set of experiments I also ran each 3 times to get an average time, with 100 million integer per iteration. Just like in the previous paragraph, the speed up on “Stack Quicksort v3.0 and v4.0” and “PThreads Quicksort v2.0” lose large performance gains at 8 threads.

Notably, the problem with speed comparisons between their speed and mine, is since we are using “-O3” flags it is difficult to see any improvements because compilers now are smart enough to deal with replacement of declared types with primitive types. Therefore, the only real comparison can be made between the versions without compiler optimization. Unfortunately their data for these versions were not provided. However, assuming that my versions are as accurately close to theirs, then we can assume the speed ups without “-O3” is definitive.

Results

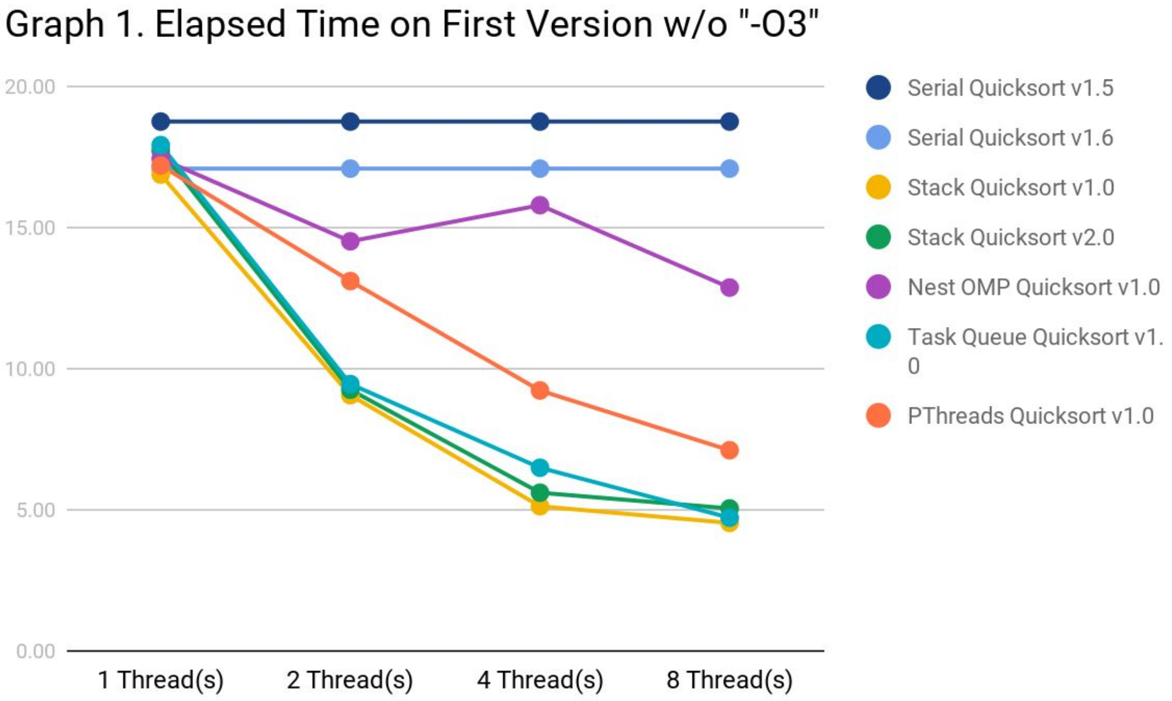

Table 1	1 Thread(s)	2 Thread(s)	4 Thread(s)	8 Thread(s)
Serial Quicksort v1.5	18.75	18.75	18.75	18.75
Serial Quicksort v1.6	17.09	17.09	17.09	17.09
Stack Quicksort v1.0	16.88	9.06	5.13	4.54
Stack Quicksort v2.0	17.72	9.25	5.61	5.05
Nest OMP Quicksort v1.0	17.44	14.51	15.79	12.87
Task Queue Quicksort v1.0	17.92	9.46	6.49	4.72
PThreads Quicksort v1.0	17.20	13.10	9.23	7.11

Graph 2. Elapsed Time on First Version w/ "-O3"

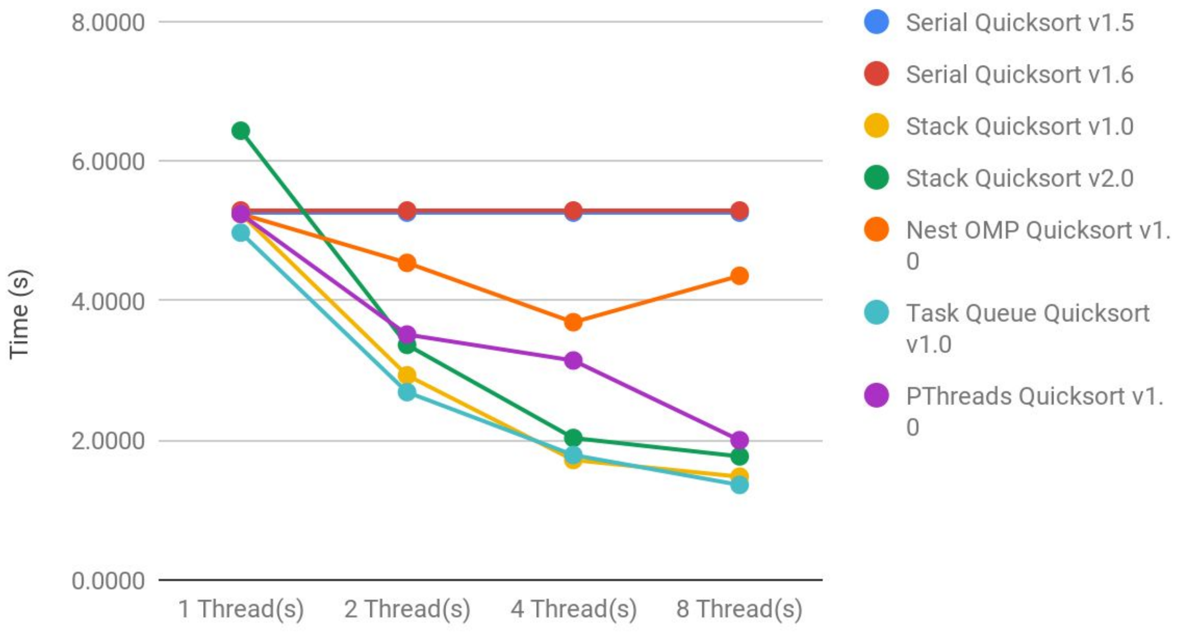

Table 2	1 Thread(s)	2 Thread(s)	4 Thread(s)	8 Thread(s)
Serial Quicksort v1.5	5.2633	5.2633	5.2633	5.2633
Serial Quicksort v1.6	5.2933	5.2933	5.2933	5.2933
Stack Quicksort v1.0	5.2473	2.9323	1.7187	1.4800
Stack Quicksort v2.0	6.4367	3.3683	2.0340	1.7707
Nest OMP Quicksort v1.0	5.2457	4.5437	3.6930	4.3580
Task Queue Quicksort v1.0	4.9767	2.6913	1.7900	1.3617
PThreads Quicksort v1.0	5.2457	3.5160	3.1447	2.0023

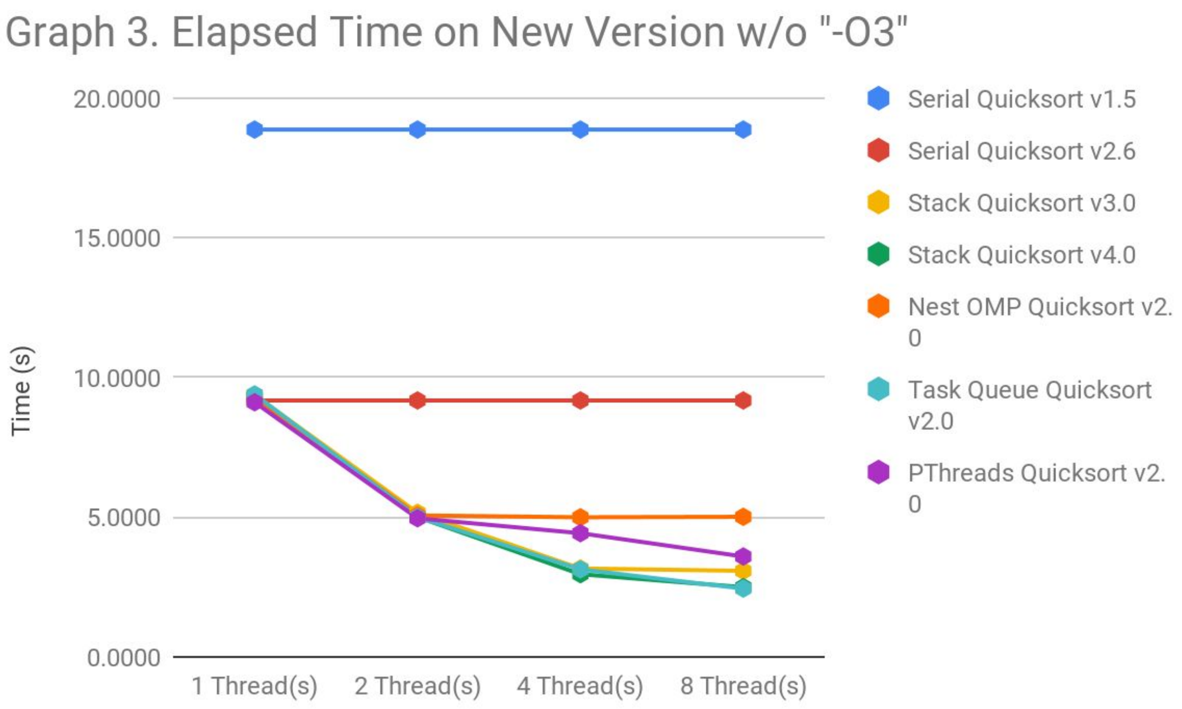

Table 3	1 Thread(s)	2 Thread(s)	4 Thread(s)	8 Thread(s)
Serial Quicksort v1.5	18.8757	18.8757	18.8757	18.8757
Serial Quicksort v2.6	9.1800	9.1800	9.1800	9.1800
Stack Quicksort v3.0	9.3743	5.1613	3.1653	3.0817
Stack Quicksort v4.0	9.3143	5.0087	2.9613	2.4973
Nest OMP Quicksort v2.0	9.2870	5.0657	4.9983	5.0160
Task Queue Quicksort v2.0	9.3960	5.0060	3.1333	2.4460
PThreads Quicksort v2.0	9.1153	4.9627	4.4293	3.5947

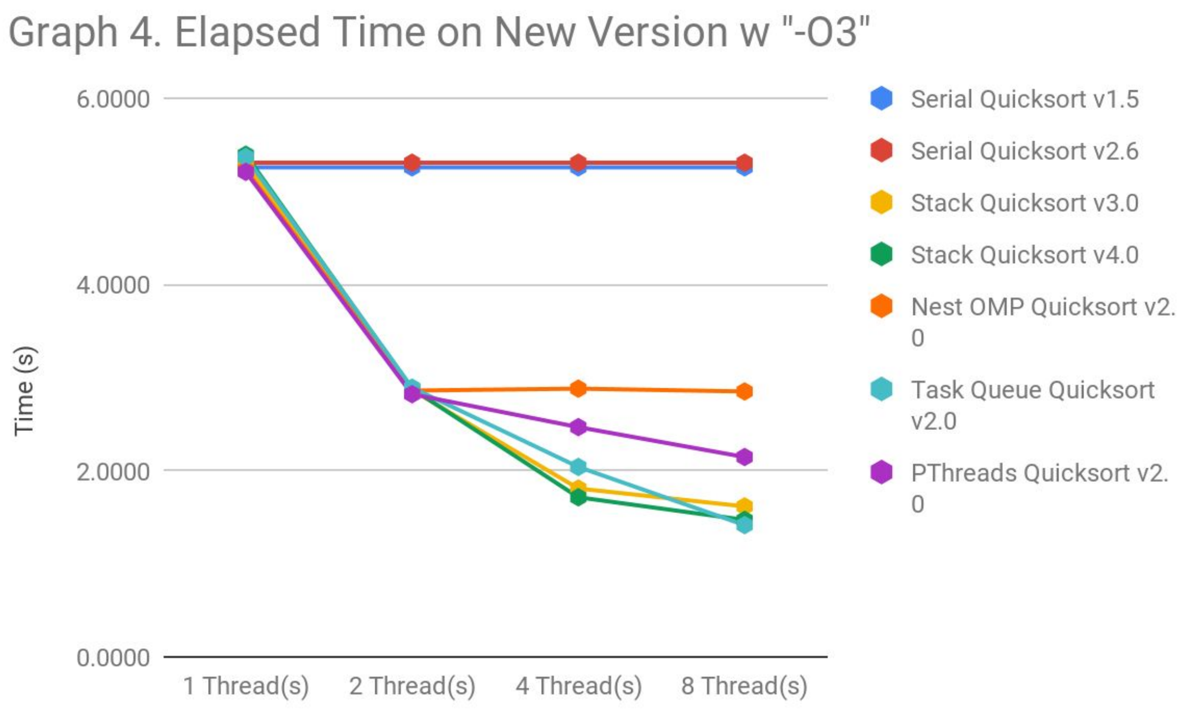
Graph 4. Elapsed Time on New Version w "-O3"

Table 4	1 Thread(s)	2 Thread(s)	4 Thread(s)	8 Thread(s)
Serial Quicksort v1.5	5.2570	5.2570	5.2570	5.2570
Serial Quicksort v2.6	5.3070	5.3070	5.3070	5.3070
Stack Quicksort v3.0	5.3047	2.8653	1.8083	1.6163
Stack Quicksort v4.0	5.3930	2.8803	1.7143	1.4713
Nest OMP Quicksort v2.0	5.2297	2.8600	2.8810	2.8507
Task Queue Quicksort v2.0	5.3680	2.8913	2.0400	1.4153
PThreads Quicksort v2.0	5.2090	2.8213	2.4677	2.1467

Hardware

Table 5. Personal Computer - CPU Intel i7-4790K, 8 threads total.

CPU	Physical Cores	Logical Cores	Speed	Registers	Threads	SMT
1	4	4	4.00GHz	64bit	8	yes

Table 6. Personal Computer - CPU Intel i7-4790K, 8 threads total.

Cores	L1	L2	L3
1-4 each	4x 32KB 8-way associative instruction caches 4x 32KB 8-way set associative data caches	4x 256KB 4-way set associative caches	6MB 12-way set associative shared cache

Then 1 physical core has a total cache of L1 256KB + L2 1024KB + L3 6144KB = 7424KB = 7.25MB.
So 4 physical cores have a total of 7.25MB*4 = 29MB.

Conclusion

There are many ways that the code of this paper can be improved by simply replacing named types with primitives, however, after compiling with "-O3" these improvements are virtually negligible, as the compiler is smart enough to do something that appears to result in the same differences. It appears that the speed up using more threads can be either a good time improvement, or appears to result in no change by adding more threads. There are possible asymptotic improvements that can be made upon my changes as well, although I'm not sure where they could be added. Lastly, its possible they are not using n a completely random array and one that results in a small computation time resulting in slightly faster computation time. I believe that my speeds would be a little better than the papers if I had access to the compiled versions without "-O3" optimization applied, and if I was using the same array that they used for sorting.

References

Süß, M., & Leopold, C. (2004, October). A user's experience with parallel sorting and OpenMP. In *Proceedings of the Sixth European Workshop on OpenMP-EWOMP'04* (pp. 23-38).