Tour d'Algorithms: OpenMP Qualifier

Group:

Brandon Hudson	800474727	brhudso@siue.edu
Sam Schrader	800544026	saschra@siue.edu
Jan-Niklas Harders	800733249	janhard@siue.edu

1. Bubblesort

We noticed that the serial bubble sort was only capable of sorting smaller input sizes in a reasonable amount of time. This algorithm had the longest running times, which is what you would expect, considered its time complexity of n^2. This project has shown that the bbs took the longest out of the sorting algorithms we evaluated. A time complexity of n^2 means that every input will be processed for the number of inputs there are. This explains its relatively long running time compared to the other algorithms we tested.

Our parallel version of bubble sort (bbp) does outperform the standard serial implementation (bbs), but still appears to be O(n^2) for the array sizes that we tested and dwarfs the C standard sort (reference) (see Figure 1). We could not finish timing bbs for the maximum size used with bbp and reference, but it was over 3600 seconds or 1 hour.

Our algorithm consists of two passes: the first is multithreaded, while the second pass is a serial insertion sort which finalizes things, as the first pass does not fully sort the data. In the first pass, every thread is given a subset of the array which it then bubble sorts; these subsets are disjoint, meaning they do not share elements (this guarantees thread safety). The subsets are chosen so that the threads share work equally – simply put, for m threads, each thread gets every mth item with an offset of 1 for each subsequent thread, e.g., for 5 threads the first thread will bubble sort the subset of indices (0, 5, 10, 15, …), the second thread will get the subset (1, 6, 11, 16, …), and so on. See the table below for a visual reference with 5 threads. The items of each color will be sorted relative to each other, but not necessarily relative to other colors. This leads to nearly sorted data where small elements are close to the beginning and large elements are close to the end for arrays of size >> m. This motivates the choice of insertion sort as a final pass, as it is O(n) for nearly sorted data.

Table 1: Bubblesort parallel

Index	0	1	2	3	4	5	6	7	8	9	10	11
Thread	0	1	2	3	4	0	1	2	3	4	0	1

Why does this outperform serial bubble sort in the cases we tested? The main improvement is that instead of items being moved one space at a time, they are moved by m spaces (the total number of threads); this is like shell sort, except the distance between compared items does not change over time. This leads to us achieving a nearly sorted array by performing parallel bubble sorts on subsets 1/m the size of the original array (which is a quadratic speedup for bubble sort), and insertion sort should have nearly linear behavior to finalize the sort. It even appears to keep up with reference for a while, but when the array sizes become too large, it becomes clear that we are still doing a lot more comparisons and swaps than the reference sort. Each thread is still performing bubble sort on a subset of the array, so if we double the array size, we also double the subset sizes, and the O(n^2) behavior of bubble sort becomes evident eventually. The parameters for this algorithm (number of

threads and array size) interact with each other to have a complex effect on performance; there is also the issue of cache coherence, where threads may become decoupled from each other and access distant portions of the array, although the times were consistent when we tried different seeds so it's not clear how strong this effect is.

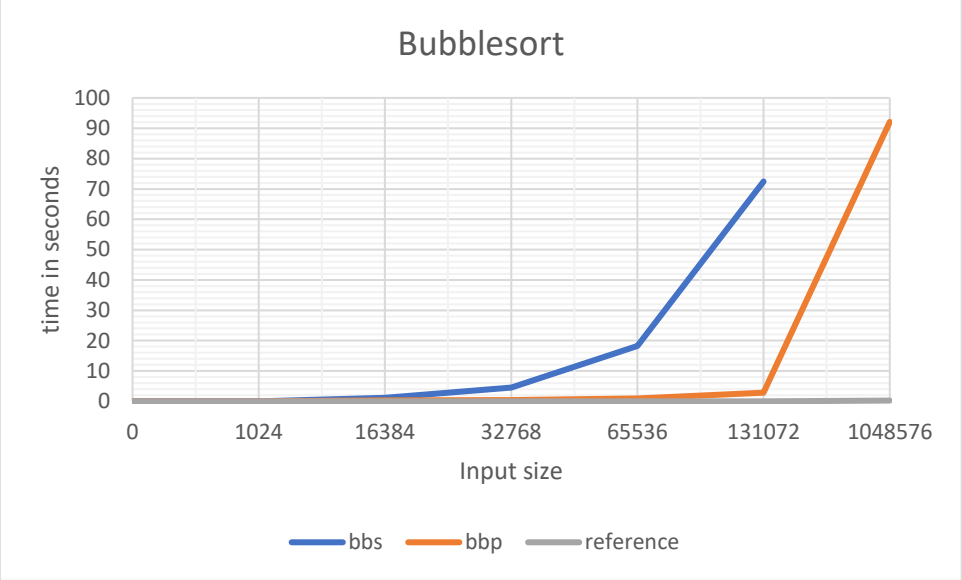

Figure 1: Recorded times for serial and parallel bubblesort algorithm in comparison to a STL sort algorithm

Table 2: Recorded times

	0	**1024**	**16384**	**32768**	**65536**	**131072**	**1048576**
bbs	0	0,0057	1,134589	4,564359	18,22061	72,45451	n/a
bbp	0	0,017297	0,287736	0,432206	0,919477	2,852933	92,09901
reference	0	0,00017	0,003434	0,007229	0,014619	0,02905	0,247124

2. Quicksort

Out of the three sort algorithms that we wrote qsp diverged at around the same larger sort size and as it approached 1.00E+06 sort size qsp managed to slightly perform the best which was surprising to see compared to the other results with the reference consistently being the fastest sort (see Figure 2). Originally, we were only going to run each sort with data sizes of {10,100,1000,10000}, but we thought our input size might be too small to get a good representation of the multithreading. We decided to run a sort with a larger sort size, and we saw that the times started to get closer the larger the size to sort. At about sort size of 1.00E+06 qsp began preforming slightly better than the single threaded sort and the reference sort. Between 1.00E+06 to 1.00E+07 qsp preformed the best on the graph sorting those size the quickest, but around this point 1.00E+07 the algorithms begin to diverge with qss clearly taking longer than the reference sort and the qsp. Eventually the reference sort was the quicker sort with the largest tested sizes, but it was interesting to see the qsp not being far behind. The difference is much closer between the reference sort and qsp than qsp to qss the larger the sort size. These tests helped us to understand better the scope of input sizes that could be sorted by modern processors. Initially our input sizes were too small and, in most cases, the single threaded out preformed the multi-threaded until larger input sizes. In real world problems you might have to sort millions even billions of items potentially and if that is the problem that you need to solve qsp could be an useful tool, but at smaller sizes it seems like the processer is fast enough to sort smaller inputs sizes with a single thread and adding more threads slows down the algorithm.

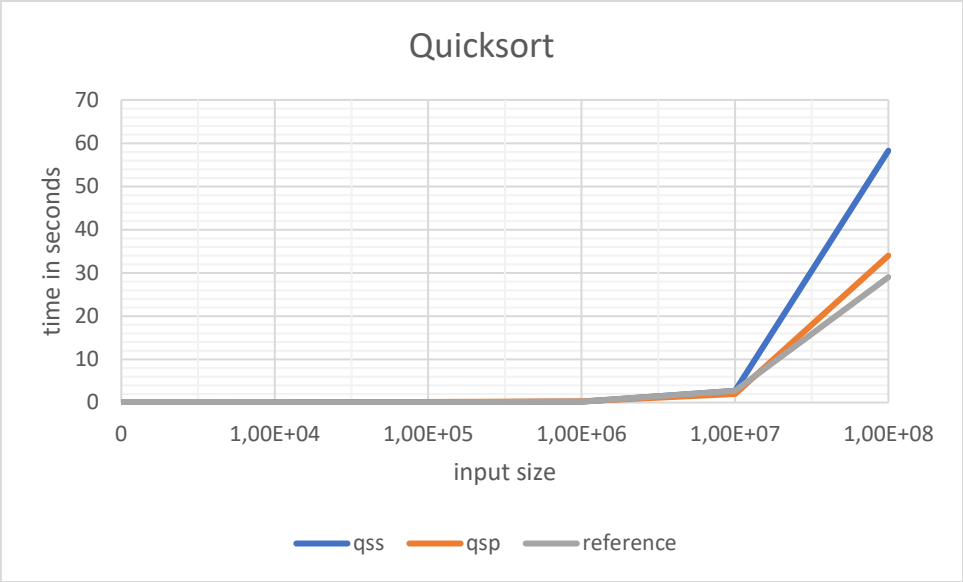

Figure 2: Recorded times for serial and parallel quicksort compared to STL sort algorithm

Table 3: Recorded times quicksort

	0	**1.00E+04**	**1.00E+05**	**1.00E+06**	**1.00E+07**	**1.00E+08**
qss	0	0,001956	0,021772	0,228473	2,73018	58,255647
qsp	0	0,034713	0,166989	0,27527	2,004866	33,999634
reference	0	0,002015	0,022133	0,236099	2,692258	29,010334

3. Mergesort

The serial merge sort is a standard implementation. It recursively splits the array in half until it is working with 1-item arrays, then calls a method which can merge two sorted arrays into a single sorted array (1-item arrays are naturally already sorted). Thus, single elements are merged into 2-item arrays, which are merged into 4-item arrays, and so on until we have a sorted version of the original array. This algorithm is O(n*log(n)) in all cases, as it greatly reduces the number of comparisons and swaps – items are only compared and swapped with nearby items (both in terms of the array and in terms of numerical distance).

The parallel merge sort (msp) unfortunately performs much worse than either the serial version (mss) or the C standard sort (reference) (see Figure 3). This is because it is a naïve implementation which parallelizes the “splitting” function calls but serializes on the merging operations. It is meant to have a parent thread which forks off a “task” thread that handles one of the recursive splits, while the parent thread handles the other; they then wait to join up before the parent thread calls merge. However, as the parallel version performs much worse (to the point of not being runnable for array sizes that mss can handle), it is likely that our implementation is not behaving correctly. It does still produce correct output, but likely is making way more function calls than it should and taking up much more stack space than the serial version, due to duplicate calls by all 8 threads. (It is strange that this would still end up producing correct output, but this is the explanation that makes the most sense to us). More sophisticated parallelizations of merge sort actually use a merging operation which can be run in parallel, leading to a more significant speedup as that is where most of the actual computation is happening with merge sort.

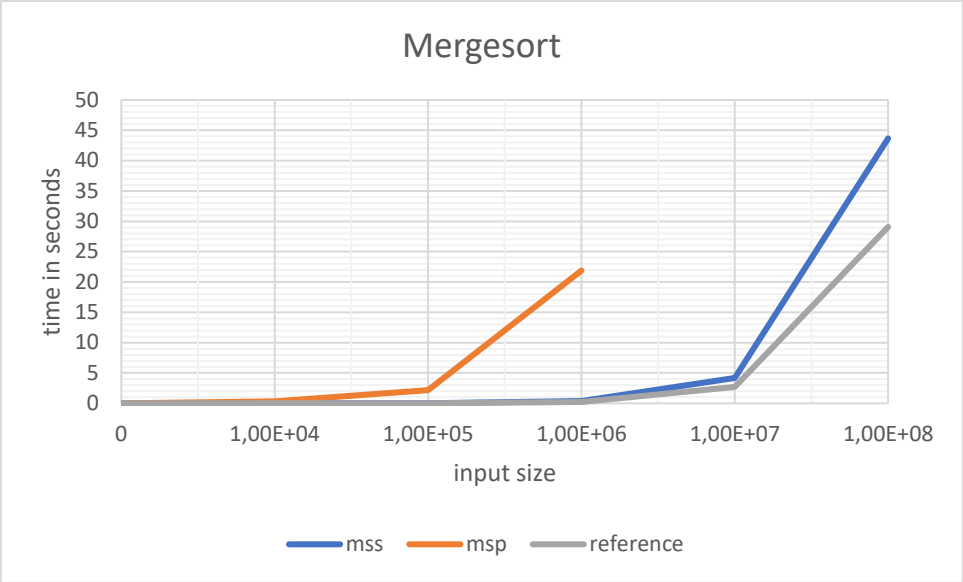

Figure 3: Recorded times for serial and parallel mergsort compared to a STL sort algorithm

Table 4: Recorded times for serial and parallel mergesort

	0	**1.00E+04**	**1.00E+05**	**1.00E+06**	**1.00E+07**	**1.00E+08**
mss	0	0,003968	0,036247	0,377774	4,165734	43,661046
msp	0	0,30682	2,170374	21,875405	n/a	n/a
reference	0	0,002022	0,022603	0,236145	2,693829	29,073562