Benchmarking parallel sorting by regular sampling

Anthony Ou
SID: 1248175
CSID: 81995
CMPUT 481
Fall - 2016

1. Introduction

This paper explores an implementation of the parallel sorting by regular sampling algorithm. The main points to determine is if the algorithm is correct. And in addition, determine if the speedup is significant when the PSRS algorithm is implemented correctly.

2. Implementation

The implementation follows closely to what was given by Li, et al. The major difference is the program execution is limited to a single machine with one shared memory space. In addition, some of the phases were changed to accommodate the implementation. Namely, phases three and four were combined into a single phase. Each core would iterate through each ith chunk and then merge into a global array for the ith processor. With this method, there is no need to add extra synchronization between phase three and four. In this implementation, phase four is a simple concatenation of the lists from all the processors. Another difference is in phase 2 where a single processor is responsible for processing the pivots. This implementation uses the main process thread to sort the regular sample list and determine pivots. The main process thread would be an additional thread to the number of worker threads.

3. Methodology

The list being sorted is a generated normal distribution from a range of 1 to $2^{32} - 1$. All the data is kept in memory and not included in the benchmark time, nor is the teardown operations of the program. The program is programmed to time the run time of the different phases and the total runtime, this allows the timings to be more granular. The lists being generated starts from 8 million to 11 million elements with an increment of 1 million. The cpu used for all tests is a i5-4690K, four core CPU, running at 3.50GHz. Additional tests were made by varying number of threads created, starting from one thread to six threads. Each run was made 5 times and averaged out.

After each run, the correctness of the program is evaluated. A standard library function is run to check if the final list is sorted, the time to run this check is not included in total run time. If the final list is sorted the program will exit with a status code of zero, otherwise the program will exit with a non-zero status code.

A sequential sorting program was written as a point of comparison for the speedup charts. This program used a sequential merge sort on the same number of elements.

4. Empirical Analysis

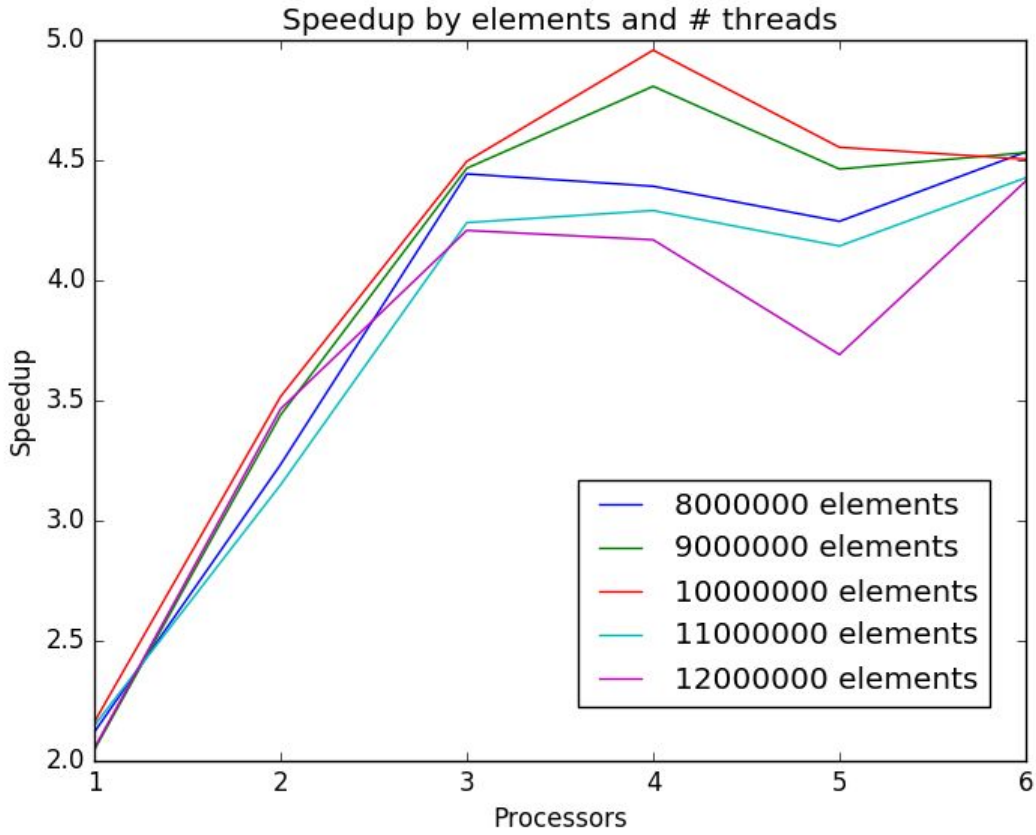

Figure 1. Graph of speedup vs the number of elements being sorted. Each line represent the number elements in the array. Done on a 4 core processor and a uniform distribution.

The biggest speedup is found when the number of threads is equal to the number of cores in the processor. These findings make sense because PSRS splits the work based on the number of threads created. The work granularity will be too fine when there are more threads than there are cores. Subsequently, the workload will be too coarse when there are fewer threads than there are cores. Using fewer threads than there are cores would be slower since the parallel portions of the code that would be run on a free core are no longer.
Another characteristic is a slight slowdown when using more threads than there are cores. What is likely happening is slowdown caused by the operating system scheduling active and sleeping threads. In addition, there might be much higher variability as the operating system schedules threads to run.
What is surprising the speedup of 2 when using even 1 thread. No pivoting or merging is done when one thread is used, so all the sorting is done in phase one.

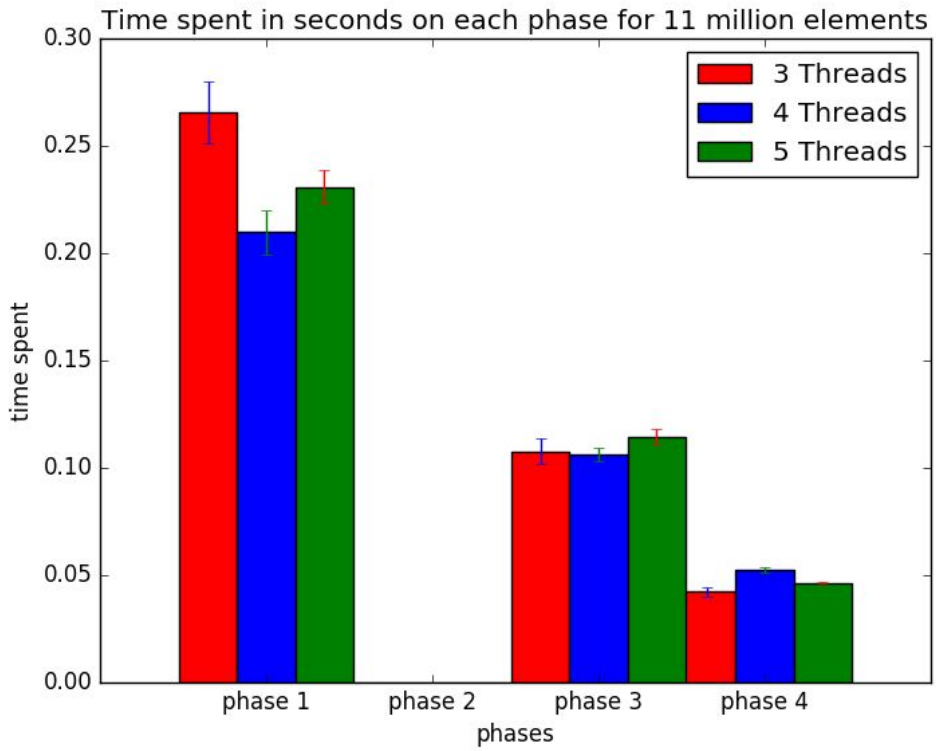

Figure 2. Time spent on each phase with a uniform distribution with various number of threads. On a 4 core CPU.

A majority of time is spent on phase 1, this is because each subarray will be unsorted. And each processor will sort the entire subarray assigned to them using a sequential quicksort algorithm. Phase two takes very little of the total execution time. Since the size of phase two only grows with the number of processors. Phase two is not a bottleneck.
There is a noticeably more time spent in phase 1 when there are fewer threads than there are cores in phase one. An explanation could be variation in the quicksort using the phase 1. However, a more likely explanation is that when four threads are created on a four core CPU, the OS will generally schedule on thread per core and useful work is done when otherwise a core would be spent idling.

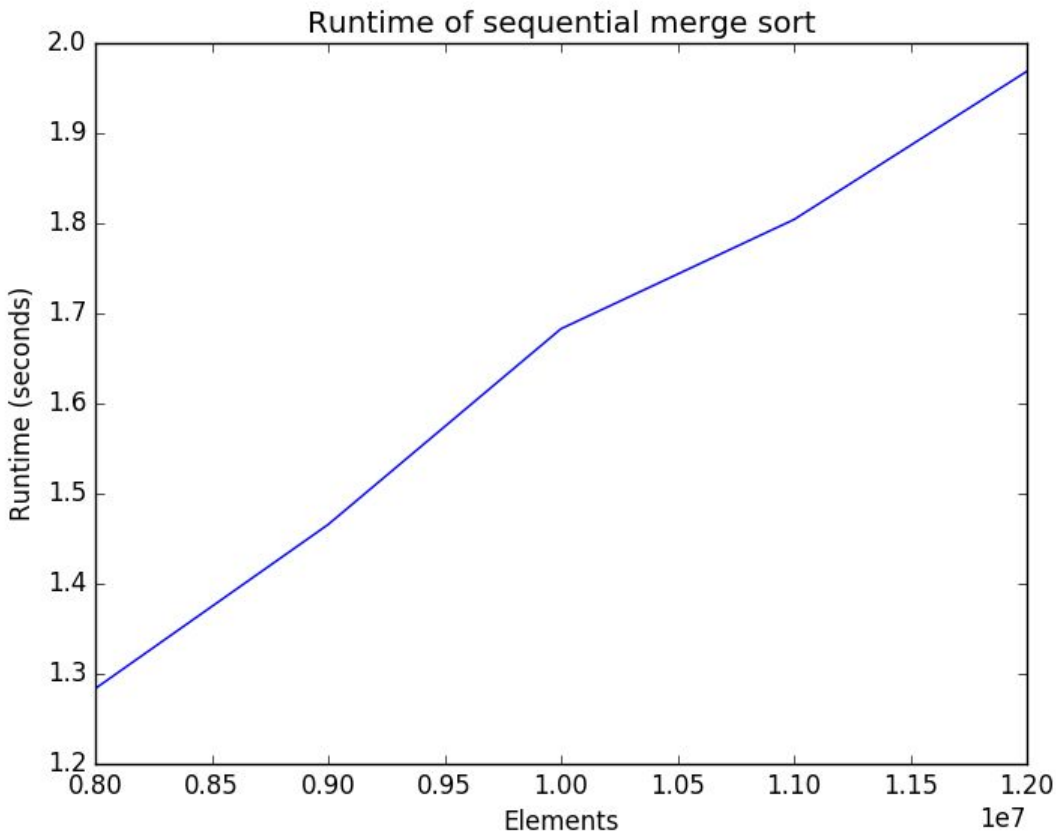

Figure 3. Runtime in seconds versus the number of elements to be sorted.

	Sorting Times in seconds				
Size	**8000000**	**9000000**	**1000000**	**11000000**	**12000000**
1 thread	**0.586046**	**0.675165**	**0.763475**	**0.827549**	**0.897816**
2 threads	**0.354354**	**0.418632**	**0.450688**	**0.49142**	**0.541684**
3 threads	**0.269177**	**0.332747**	**0.406029**	**0.415175**	**0.452935**
4 threads	**0.256014**	**0.309174**	**0.341163**	**0.35618**	**0.399799**
5 threads	**0.274825**	**0.311656**	**0.350044**	**0.389026**	**0.427089**
6 threads	**0.276941**	**0.309044**	**0.345278**	**0.378538**	**0.414109**

Table 1. Sorting times with various number of threads and elements using a uniform distribution.

Size	**8000000**	**9000000**	**10000000**	**11000000**	**12000000**
Time in seconds	**1.28612**	**1.503333**	**1.63683**	**1.818941**	**1.954017**

Table 2. Sorting times of sequential merge sort.

Sorting times with sequential merge sort increases linearly with increased input size. This is to be expected, as opposed to the PSRS algorithm.

5. Conclusion

PSRS is correct when using only one processor or multiple processors. The speedup likely to be logarithmic when the number of processors reaches infinity. Much of the execution time is spent in phases 1,3, and 4. So any improvements in the future should focus on these phases. More speedups with much large datasets using a PSRS-PSRS method where a cluster of machines is assigned a section of a list to sort and the machine itself will apply PSRS on the section.

6. References

1. **Li, X. *On the Versatility of Parallel Sorting by Regular Sampling.* Edmonton, Alta., Canada: University of Alberta, Dept. of Computing Science, 1991.**

7. Code references

1. **https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Merge_sort**
2. **https://stackoverflow.com/questions/865668/how-to-parse-command-line-arguments-in-c**