Evaluating Parallel Processing Using Merge Sort

Sam Bostian, Michael Rizig, Jonathan Turner, Eli Headley, Charlie McLarty, Ernesto Perez and Daron Pracharn
Instructor - Dr. Patrick Bobbie
Kennesaw State University
Marietta, GA
sbostian@students.kennesaw.edu

***Abstract*—Multicore processing offers the advantage of dividing and sharing computer resources among interconnected processes, mitigating bottlenecks and minimizing wasted potential caused by idle computing hardware. Given the substantial computational demands of such problems, parallelizing and distributing computing tasks across multiple cores is often more cost-effective than relying on a single powerful processor. However, one drawback of multicore processing lies in the complexity of coordinating computer resources. The objective of this project is to leverage parallelization to sort data using an implementation of Merge Sort. The approach for this project involved establishing a multithread pool and utilizing the Single Program Multiple Data (SPMD) model. Comparisons were made with the speedup, efficiency, and runtimes achieved by increasing the number of distributed cores across different array sizes against the metrics of a single-core processor.**

Keywords—Threading, Multithreading, Merge sort, Parallel Computing

Introduction

This experiment aims to answer the question: "How significant would the performance increase if the datasets become exponentially larger for each increase in the number of threads used for processing?" An attempt to answer this question by increasing each of the data sets using arrays with sizes of 10,000, 100,000, 200,000, and 300,000. Each array was populated with random and distinct integers ranging from 1 to 999,999. Subsequently, the arrays were transmitted via a master thread to a thread pool. The thread pool then executes a merge sort on the divided components on the array then passes the results to the master thread to reassemble the array. The results obtained for each scaled array size were compiled into a table and graphed to analyze the results.

Data Analysis

From each experiment the time was collected, in nanoseconds. The timing begins at the creation of the threads and finishes when the data from each thread finishes merging creating a complete sorted array. For the experiments the improvement in performance of increasing the number of parallel processors versus the serial one, is measured using the speedup and efficiency metrics. The speedup, the ratio of the program runtime in serial over the runtime in parallel:

$$Speedup(n,p) = \frac{T_{Serial(n)}}{T_{Parallel(n,p)}} \quad [1]$$

Where n is the size of the input and p is the number of processors. A perfect speedup score is where the speedup equals the number of processors, Speedup(n,p) = p, also known as linear speedup. To determine how each processor contributed to the speedup parallel efficiency is used. Parallel efficiency is calculated using the following formula:

$$Efficiency(n,p) = \frac{Speedup(n,p)}{p} = \frac{T_{Serial(n)}}{p * T_{Parallel(n,p)}} \quad [1]$$

Parallel efficiency is given by the speedup over the number of processors.

Results

As the array size increased the benefits of parallel processing can be seen in Table 1. When the array is only 10,000 the time it takes to process the array is approximately half from one to two and from two to four processers. When eight and sixteen processors are used on an array of 10,000 elements the decrease in time to process the arrays is hampered by the work to divide the array. The runtime decreases between about three quarters when the number of processors is increased between two and four processors. When the processors are increased to eight or sixteen the runtime is only decreased by about half.

Runtime				
Number of Threads	Array Sizes (# of elements)			
	10000	100000	200000	300000
p = 1	591	48221	183908	403233
p = 2	258	14191	57020	109616
p = 4	123	4532	15974	34549
p = 8	123	2278	8335	18565
p = 16	104	1257	4668	10450

Table 1: Runtimes for the randomly generated arrays.

While an array of 10,000 elements might look like it does not decrease but that is due to the fact that the 300,000 elements decrease so much compared to the 10,000 elements array.

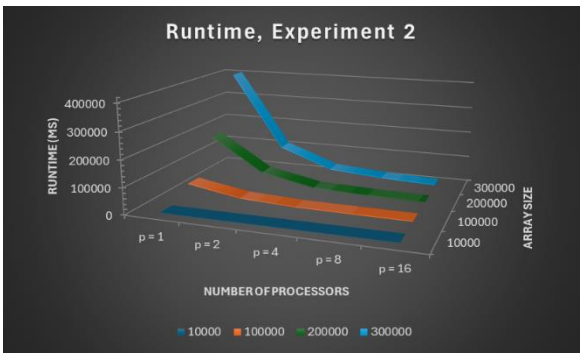

Figure 1: A line graph displaying the recorded runtimes for each array size for a given number of processors.

The speedup gained by using parallel processing is almost equal for each processor regardless of the size of the array. The speed up almost increase by the same factor has the number of processors doubles. The speed up is more than tripled from one processor to two processors and from two to four processors. The speedup increases of a factor of two when the number of processors increases from four to eight and then from eight to sixteen the speedup is only gained by a factor of 1.75.

Speedup				
Number of Threads	Array Sizes (# of elements)			
	10000	100000	200000	300000
p = 1	1	1	1	1
p = 2	3.68	3.4	3.23	3.68
p = 4	11.67	10.64	11.51	11.67
p = 8	21.72	21.17	22.06	21.72
p = 16	38.59	38.36	39.4	38.59

Table 2: Calculated speedup for the randomly generated arrays.

The graph in figure 2 shows that the speedup follows the same shape as the theoretical big O trajectory as a merge sort, which is $O(n\log_2 n)$ [2].

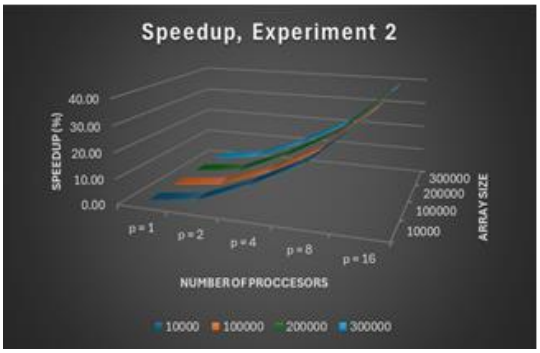

Figure 2: A line graph displaying the calculated speedup for each array size for a given number of processors.

Unlike the speedup the efficiency did increase the same amount for about all the array sizes. The 10,000-element array followed the same pattern as the larger arrays but had a lower increase in efficiency for each increase in the number of processors used. All the arrays had their greatest efficiency when four threads were used to sort an array. The efficiency for each processor begins to decrease after four threads are used.

Efficiency				
Number of Threads	Array Sizes (# of elements)			
	10000	100000	200000	300000
p = 1	1	1	1	1
p = 2	1.15	1.7	1.61	1.84
p = 4	1.2	2.66	2.88	2.92
p = 8	0.6	2.65	2.76	2.72
p = 16	0.36	2.4	2.46	2.41

Table 3: Calculated efficiency for the randomly generated arrays.

The changes in efficiency for each array are visualized in figure 3. The arrays 100,000 elements and larger the slope of the graph increases between a factor of 0.6 to 0.8 for one to eight processors used. When the number of processors is increased to eight and higher the efficiency is between 88% to 95% the efficiency of the previous number of processors. The ten thousand element array has the smallest gain in efficiency and has the largest decrease in efficiency as the number of processors increases after four threads. Efficiency only increases by 15% and then 5% as the number of processors is doubled from one to four. When the number of processors is doubled again to eight and sixteen the efficiency decreases by about 50% for each increase.

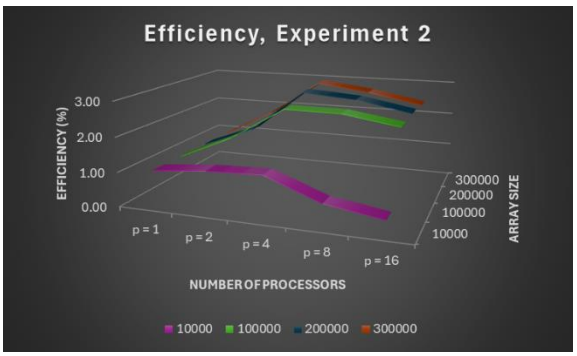

Figure 3: A line graph displaying the calculated efficiency for each array size for a given number of processors.

Conclusion

The primary goal of this research paper was to simulate parallel processing. Using SPMD task parallelism method of implementation effectively addressed the issues of multi-threading including thread synchronization and load balancing of the data among the concurrently running threads [1].

For the large arrays (>100,000), it was observed that a significant speedup occurred when the number of threads increased. This can be explained due to the cost of overhead minimal compared to the increased efficiency of the added cores. The efficiency in these large test cases indicated a speedup where the speedup is greater than anticipated for an increased number of cores.

This research showed that there is no optimal number of cores that will suit all cases. To allow more consistent results in a dynamic environment of differing input sizes, a threshold could've been implemented to assign the number of cores on runtime. Overall, it was determined that spending the extra time to implement parallel processing for a sorting algorithm yielded significantly better results when the amount of data is substantial.

References

[1] P. Pacheco, "An Introduction to Parallel Computing", Elsevier Inc., 2011. ISBN-10: 01237426095

[2] A. Levitin, "The Design and Analysis of Algorithms", Pearson, 2012. ISBN-10: 0-13-231681-1