ENSC 351 - Lab 3: MapReduce

- Lab Report -

Galen Elfert, Nic Klaassen, Diane Wolf

October 17, 2018

1 Explanation of Workload

The workload we designed as a better fit for the MapReduce framework is a distributed merge sort.

1.1 Conception

In the word count implementation, the highly-parallelised map function does virtually no work, and the overhead of spinning up threads and copying the inputs/outputs far outweighs any possible performance benefit. Looking for a better workload, we wanted to find an algorithm where a large amount of computation needs to be completed, and that computation can be broken down into discrete parts that can be completed in parallel. We also wanted to find an algorithm with a good use for the reduce stage, where some non-trivial amount of work needs to be done combining outputs from the map stage. Sorting was chosen as a better alternative to word counts, because it is a somewhat compute-heavy problem that can be easily broken down into chunks that need to be merged together.

For our distributed merge sort implementation, an array of size n is broken up into 4 chunks of size $n/4$, which are all sorted in parallel in the map stage. In the reduce stage, pairs of sorted chunks of the original array are merged into 2 chunks of size $n/2$, these 2 merges are also completed in parallel. A potential bottleneck in this implementation is the output stage, which must do a final merge of the last 2 chunks of size $n/2$.

In the limit, this algorithm can theoritically achieve a 4x speedup over a traditional merge sort based on equation 1.

$$\lim_{n\to\infty} \frac{\frac{n\log n - 2n}{4} + \frac{n}{2} + n}{n \log n} = \frac{1}{4} \tag{1}$$

Intuitively, only $O(n)$ work needs to be done sequentially in the 2 merge steps, while the rest of the $O(n \log n)$ sort is done concurrently on 4 cores.

1.2 Speed Comparison vs `std::sort`

In practice, our algorithm achieved a 2.13x speedup over a single-threaded `std::sort`, on arrays of size 100,000 to 400,000. While somewhat less than predicted, this level of speedup is fairly good considering the degree of optimization in `std::sort` and the typical losses involved with threading. Timing data for our MapReduce sort vs `std::sort` on various array sizes is shown in figure 1. Times are averaged over 100 runs.

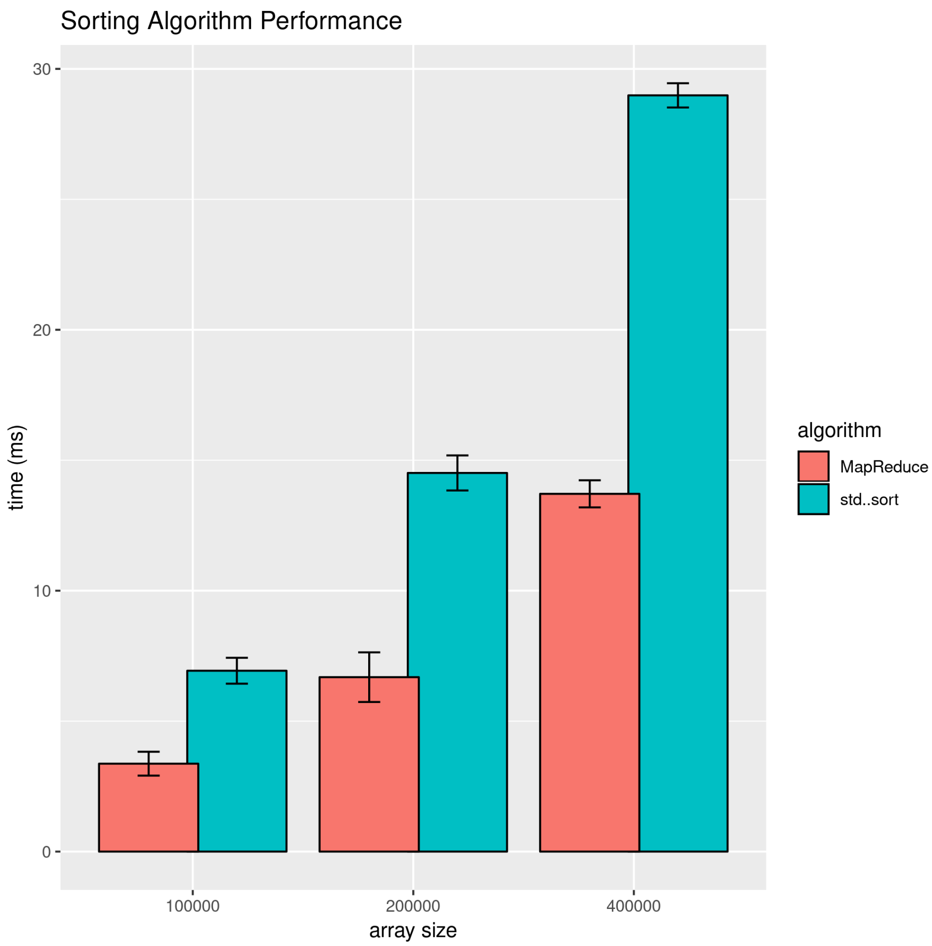

Figure 1: MapReduce Sorting vs std::sort

1.3 Compared with Word Count

The main reason this algorithm is a better fit for MapReduce than word count is that it does much more of its work in the map and reduce stages, which are executed in parallel. We analyzed this behaviour in callgrind and found that our MapReduce sorting algorithm does over 85% of its computation in the map stage. A complete breakdown is shown in table 1.

	input stage	map stage	reduce stage	output stage
word count	11.28%	11.72%	0.47%	11.21%
sort	0.00%	85.71%	5.30%	4.76%

Table 1: Word Count vs Sorting

2 Word count efficiency

Both implementations of the program counted the instances of unique words (including capitalization and adjacent punctuation marks) in fifty paragraphs of Lorem Ipsum, which is 2261 words in length. They were run on the same machine with hardware to support twelve threads. Ten executions of each following implementations were conducted, with the duration and CPU usage measured with the built-in Linux `time` command:

- single-threaded
- MapReduce, 4 threads
- MapReduce, 12 threads

Additionally, call graphs for the single-threaded and MapReduce implementations were generated using Valgrind's Callgrind tool and visualized using KCachegrind.

2.1 Single-threaded implementation

Table 2 below shows the execution time, as well as CPU usage, for each of the ten single-threaded word count runs. The word count program ran for a mean user time of 0.0016 seconds and a mean system time of 0.0028 - almost twice as long. This indicates that a great deal more time was spent by the CPU executing calls than in the program itself, which is consistent with single-threading. Since only one thread is tasked with carrying out many instructions, more time must be spent processing.

For a graphical representation of the call map for the single-threaded word count, see figure 2 at the end of this document.

Execution times for single-threaded word count				
run #	user (s)	system (s)	wall (s)	CPU usage (%)
1	0	0.004	0	0
2	0	0.008	0.01	0
3	0.004	0	0.02	0
4	0	0.004	0.01	0
5	0	0.004	0.01	0
6	0.004	0	0.01	0
7	0	0.004	0.02	0
8	0.004	0	0.01	0
9	0.004	0	0.04	0
10	0	0.004	0.01	0
mean (s)	0.0016	0.0028	0.0140	0
std. dev. (s)	0.0021	0.0027	0.0107	0

Table 2: Duration of single-threaded implementation measured by `time`

2.2 MapReduce implementation

The MapReduce implementation of the word count was tested with four threads and then the full twelve threads the machine was capable of supporting. Tables 3 and 4 below show the execution time, as well as CPU usage, for each of the ten word counts run with MapReduce with four and twelve threads, respectively. Note that the greater the quantity of threads used to multithread, the slower the program execution becomes. Further, as the thread count increased, the CPU usage also appeared to increase, going from an average of 10% with four threads to an average of 40% with 12 threads.

The four- and twelve-threaded MapReduce implementation timings both showed a greater user time compared to system time, on average. With four threads, the mean user time was 0.0084 seconds and the mean system time was 0 seconds. With twelve threads, the mean user time was 0.0104 seconds and the mean system time was 0.0040 seconds. This meant most of the time was spent in the program, not the CPU as was the case with single-threading.

For a graphical representation of the call map for the MapReduce word count, see figure 3 at the end of this document.

Execution times for MapReduce word count - 4 threads				
run #	user (s)	system (s)	wall (s)	CPU usage (%)
1	0.008	0	0.02	0
2	0.008	0	0.03	0
3	0.008	0	0.02	0
4	0.008	0	0.02	0
5	0.008	0	0.01	0
6	0.008	0	0.02	0
7	0.008	0	0.01	0
8	0.008	0	0.02	0
9	0.008	0	0.03	0
10	0.012	0	0.01	100
mean (s)	0.0084	0	0.0190	10.0000
std. dev. (s)	0.0013	0	0.0074	31.6228

Table 3: Duration of MapReduce implementation measured by `time`, with four threads

2.3 Comparison

An increase in threads used in the implementation of word counts appears to correspond with a decrease in time spent performing CPU calls and an increase in time spent in the program itself. As for wall time, there was not a clear difference between implementations - the single-threaded implementation came in with a mean wall time of 0.0140 seconds, the four-threaded MapReduce implementation with a mean wall time of 0.0190 seconds, and the twelve-threaded MapReduce implementation with a mean wall time of 0.0120 seconds.

Execution times for MapReduce word count - 12 threads				
run #	user (s)	system (s)	wall (s)	CPU usage (%)
1	0.008	0.004	0.01	0
2	0.008	0.008	0.03	0
3	0.008	0.008	0.01	0
4	0.008	0.004	0.01	0
5	0.016	0	0.01	100
6	0.008	0.004	0.01	0
7	0.012	0.004	0.01	100
8	0.008	0.008	0.01	0
9	0.012	0	0.01	100
10	0.016	0	0.01	100
mean (s)	0.0104	0.0040	0.0120	40.0000
std. dev. (s)	0.0034	0.0033	0.0063	51.6398

Table 4: Duration of MapReduce implementation measured by `time`, with twelve threads

3 Alternate Workload

In addition to the sort program, a matrix multiplication algorithm was also adapted to the MapReduce interface . The algorithm is described in detail in the blog post "Matrix Multiplication with MapReduce"[1]

The algorithm is intended for multiplying very large, sparse matrices (hundreds of thousands or millions or rows and columns) and distributing the task over many machines. Because of the permutative nature of matrix multiplication this requires significant duplication of data. The mapper stage copies every element of A for every column in B and every element in B for every row in A. Mapreduce makes this copying necessary because each worker in the reduce stage has access only to it's individual piece of the data.

On a single machine, in a multithreaded context, the mapreduce matrix multiplication operation took 100 times as long as the implementation found in the NumPy numerical computation package for Python, for a 100x100 matrix. Naturally, it also consumed significantly more memory.

In a distributed system, the workers cannot share memory, so the copying done by the mapreduce method is likely a lower bound on the amount of copying necessary for any distributed method. Since the workers are responsible for the copying, each value only needs to be transmitted once in the input stage.

4 Most appropriate workload for MapReduce

Our investigation confirmed that MapReduce is most suitable for applications where:

- The input data can be partitioned into discrete pieces which can be operated on in isolation.

[1] https://lendap.wordpress.com/2015/02/16/matrix-multiplication-with-mapreduce/

- The computation required for each piece is significant enough to overcome the overhead of partitioning, distributing and then recombining the data.

5 Impact of using multiple machines on execution speed

The primary difference between a multi-machine implementation and a local, multi-threaded implementation is that the overhead of moving data during the input, shuffle, and output stages is much higher when moving data between machines, versus merely moving data around in local memory. This increases the importance of the suitability criteria described above.

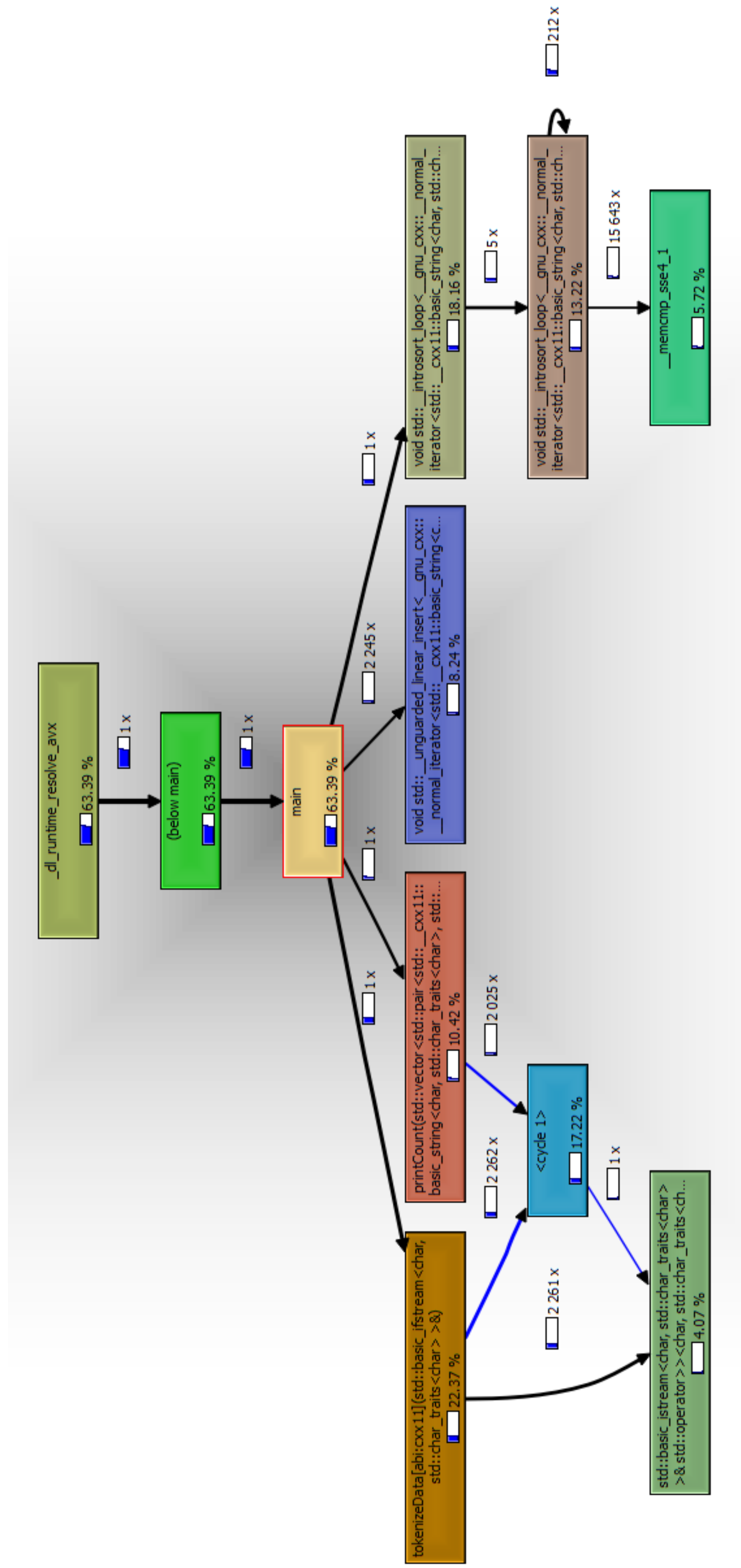

Figure 2: Call map for single-threaded implementation of word count

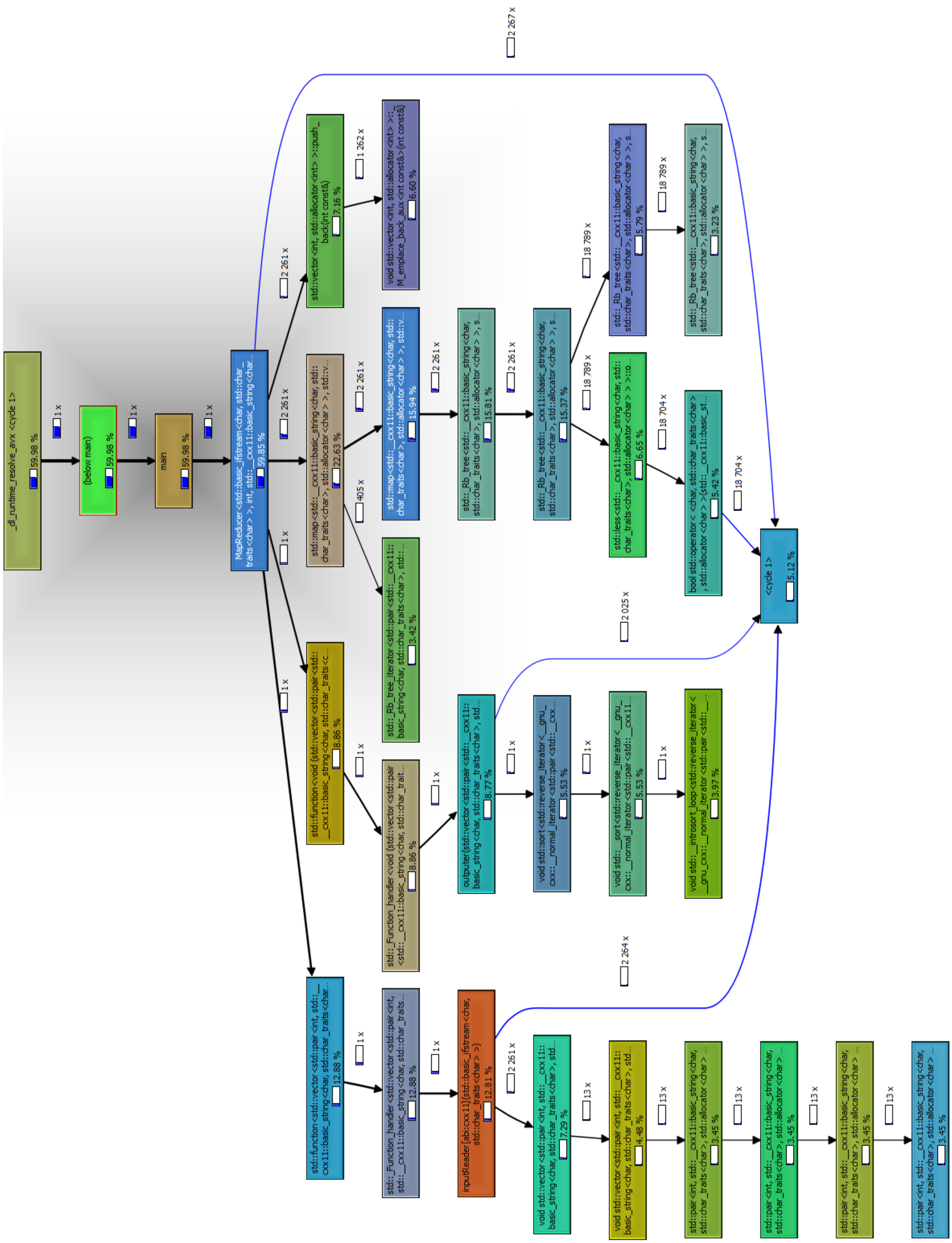

Figure 3: Call map for MapReduce implementation of word count (twelve threads)