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.

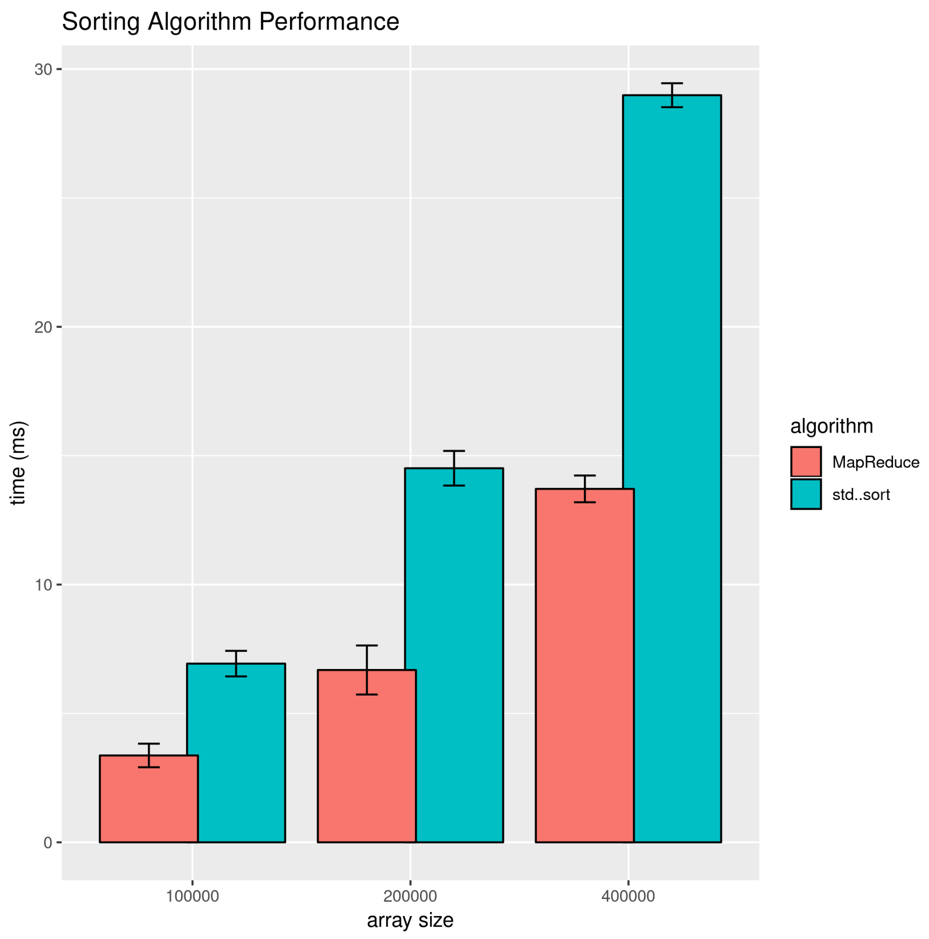

Figure 1: MapReduce Sorting vs std::sort

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.

2 Word count efficiency

Both implementations of the program counted the instances of words in fifty paragraphs (with a total length of 2261 words) of Lorem Ipsum. They were run on the same machine with hardware to support twelve threads. Ten executions of each implementation were conducted,

	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

with the duration measured by the built-in Linux `time` command. A call graph for both implementations was generated using Valgrind's Callgrind tool.

2.1 Single-threaded implementation

The single-threaded implementation of the word count ran for a mean wall time of 0.0140 seconds. Table 1 below shows the execution time, as well as CPU usage, for each of the ten single-threaded word count runs.

See Figure 2 for a graphical representation of the call map.

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 2 and 3 below show the execution time, as well as CPU usage, for each of the ten word counts run with MapReduce. Note that the greater the quantity of threads used to multithread, the slower the program execution became. 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.

See Figure 2 for a graphical representation of the call map.

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

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

2.3 Comparison

3 Most appropriate workload for MapReduce

Data that needs sorting?

4 Impact of using multiple machines on execution speed

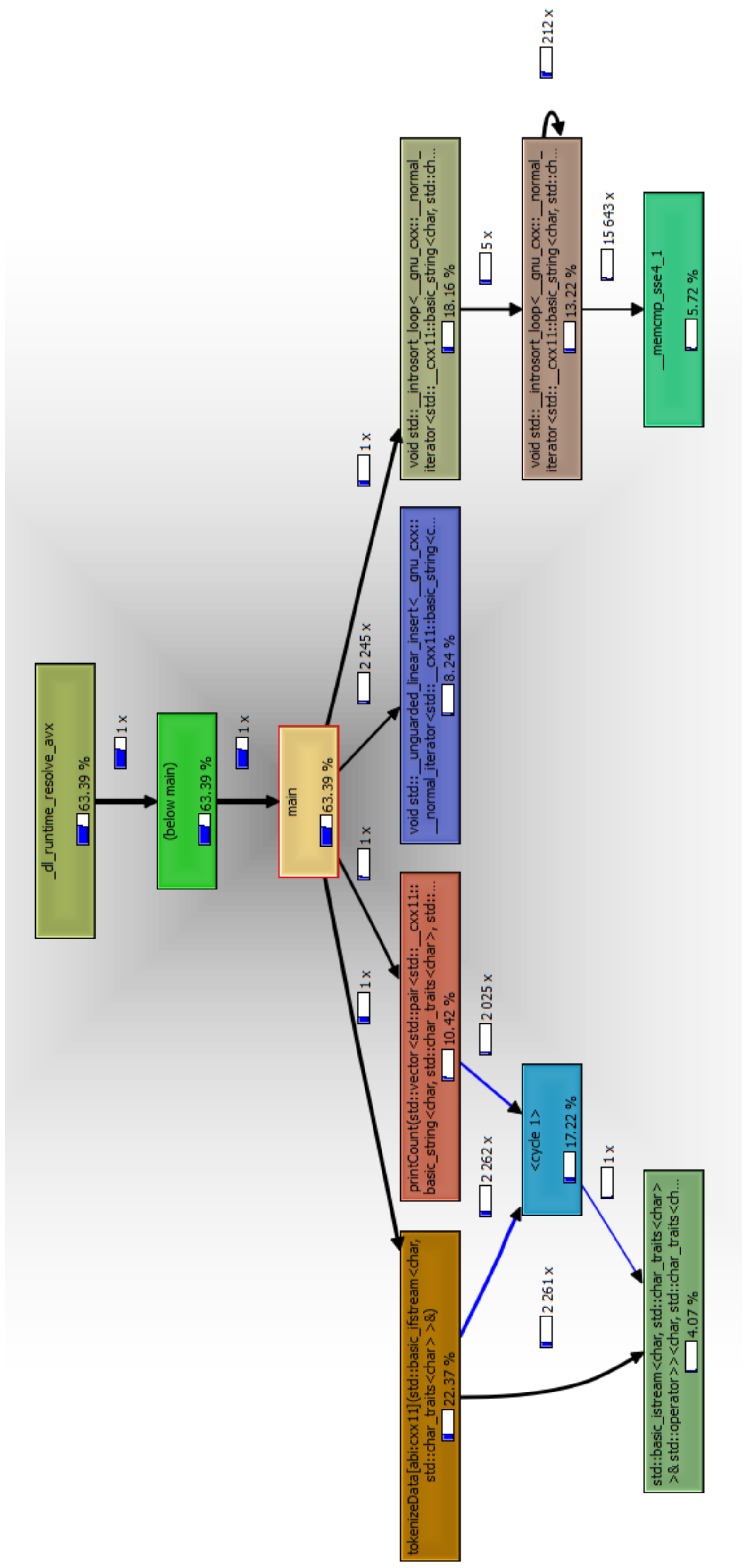
_dl_runtime_resolve_avx
63.39 %
1 x
(below main)
63.39 %
1 x
main
63.39 %
1 x
void std::__introsort_loop<__gnu_cxx::__normal_iterator<std::__cxx11::basic_string<char, std::ch...
18.16 %
5 x
void std::__introsort_loop<__gnu_cxx::__normal_iterator<std::__cxx11::basic_string<char, std::ch...
13.22 %
212 x
15 643 x
__memcmp_sse4_1
5.72 %
2 245 x
void std::__unguarded_linear_insert<__gnu_cxx::__normal_iterator<std::__cxx11::basic_string<c...
8.24 %
1 x
printCount(std::vector<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::...
10.42 %
2 025 x
<cycle 1>
17.22 %
1 x
1 x
tokenizeData[abi:cxx11](std::basic_ifstream<char, std::char_traits<char> >&)
22.37 %
2 262 x
2 261 x
std::basic_istream<char, std::char_traits<char> >& std::operator>><char, std::char_traits<ch...
4.07 %

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

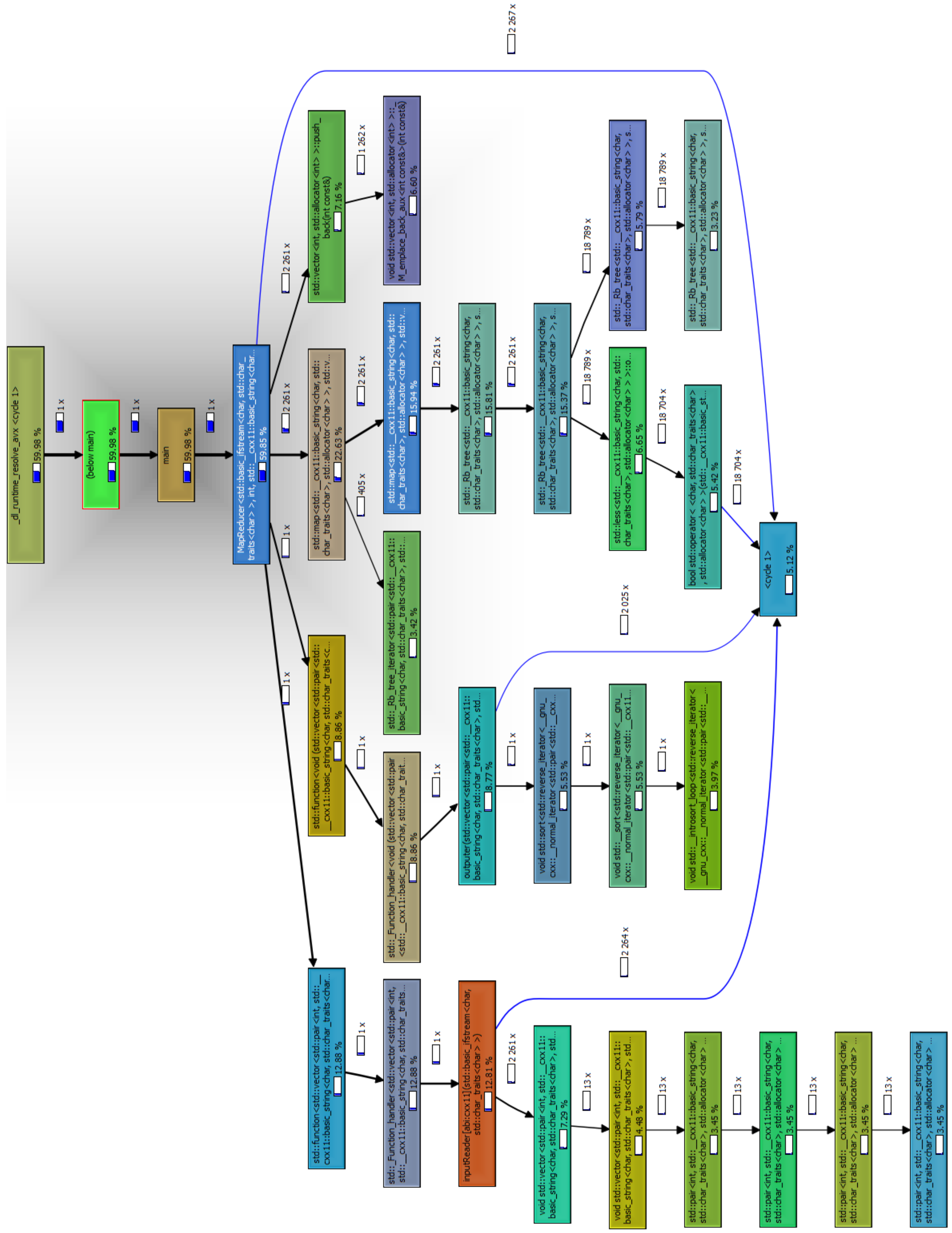
_dl_runtime_resolve_avx <cycle 1> 59.98 %
(below main) 59.98 %
main 59.98 %
MapReducer<std::basic_ifstream<char, std::char_traits<char> >, int, std::__cxx11::basic_string<char... 59.85 %
std::vector<int, std::allocator<int> >::push_back(int const&) 7.16 %
void std::vector<int, std::allocator<int> >::_M_emplace_back_aux<int const&>(int const&) 6.60 %
std::map<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::v... 22.63 %
std::map<std::__cxx11::basic_string<char, std::char_traits<char> >, std::v... 15.94 %
std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, s... 15.81 %
std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, s... 15.37 %
std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, s... 5.79 %
std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, s... 3.23 %
std::less<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::o... 6.65 %
bool std::operator< <char, std::char_traits<char>, std::allocator<char> >(std::__cxx11::basic_st... 5.42 %
<cycle 1> 5.12 %
std::_Rb_tree_iterator<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::... 3.42 %
std::function<void (std::vector<std::pair<std::__cxx11::basic_string<char, std::char_traits<c... 8.86 %
std::_Function_handler<void (std::vector<std::pair<std::__cxx11::basic_string<char, std::char_trait... 8.86 %
outputer(std::vector<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std... 8.77 %
void std::sort<std::reverse_iterator<__gnu_cxx::__normal_iterator<std::pair<std::__cxx... 5.53 %
void std::__sort<std::reverse_iterator<__gnu_cxx::__normal_iterator<std::pair<std::__cxx11... 5.53 %
void std::__introsort_loop<std::reverse_iterator<__gnu_cxx::__normal_iterator<std::pair<std::__... 3.97 %
std::function<std::vector<std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char... 12.88 %
std::_Function_handler<std::vector<std::pair<int, std::__cxx11::basic_string<char, std::char_traits... 12.88 %
inputReader[abi:cxx11](std::basic_ifstream<char, std::char_traits<char> >) 12.81 %
void std::vector<std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std... 7.29 %
void std::vector<std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std... 4.48 %
std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> ... 3.45 %
std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> ... 3.45 %
std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> ... 3.45 %
std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> ... 3.45 %
1 x
2 261 x
2 267 x
1 262 x
405 x
18 789 x
18 704 x
2 025 x
2 264 x
13 x

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