ECE 563

Songrui Li 0025338817

Taojun Wang 0029203447

04/25/2018

Large Project Report

I. Summary

This report performs analysis and discussions based on the project requirement which is to implement iteration of a parallelized word count algorithm using both OpenMP and MPI method. The OpenMP version was implemented to support a shared memory model. The program should be able to calculate word count in some text files. To achieve that, a global hash table was initialized for every word appeared in text files and made additions to each counter every time the same word appeared again. The hash table would produce an output file that contains word count for every word and the elapsed time. Both versions of the program contain four parts: reader, mapper, reducer and writer. Each version utilizes reader threads to read data from files, mapper threads to generate a local word count, reducer threads to merge word counts across threads/nodes and finally writer threads to return the results. The MPI version has some sender and receiver threads lie in between the parts compare to the OpenMP version.

After comparing the output results from both versions, the OpenMP version performed better and had more speedup than the MPI version. The difference between

the two versions was the way communication was orchestrated. There can only be one MPI communication from a given node at a time for the MPI version. And threads needed to do more communications in MPI and would have more parallel overhead due to the communication. Moreover, queuing is simpler and faster when using OpenMP because of the nature of sharing data between cores. More details and performance metrics will be performed below.

II. Algorithm Analysis

i) OpenMP version procedure and analysis

a) Worker thread reads the text files (reader)

b) Worker thread splits the sentences and puts each word into the hash table with the (key, value) pair as (word, number of appearance) (mapper)

c) Master thread does the reduce, count through all the hash tables and then return the result (reducer)

d) Master thread writes the result to an output file (writer)

OpenMP allows communication from any node at any time. Since the communication can easily be done through memory. All the threads should work on one problem and then move on to the next one. The reader and mapper parts are performed by the worker thread. Therefore, they are the parallel part that may have speedups. But the reducer and writer parts cannot be parallelized, they are performed by only one master thread

sequentially. Thus, with large input to perform, the performance bottleneck would be related to the speed of implementing the sequential reducer and writer part.

ii) MPI version procedure and analysis

a) Root node reads the text files (reader)

b) Root node sends Bcast of the input to worker nodes

c) Worker nodes split the sentences and do the hash table input as the OpenMP version (mapper)

d) Each node sends its hash table to the next node and the last node sends it back to the root node (reducer)

e) Root node writes the result to an output file (writer)

The MPI version which implemented to read all the files by different nodes at the beginning was failed because of large workload and small number of nodes. Since only one MPI communication operation could be performed at a time. To prevent conflicts during the message passing, locks is helpful when implementing the mapper and reducer. One node is used to do the reader things and sent out the data to other nodes by message passing. Reducer and writer are also performed sequentially by one node same as the OpenMP version. The performance bottleneck is related to the sequential reader, reducer and writer in the MPI version.

III. Results

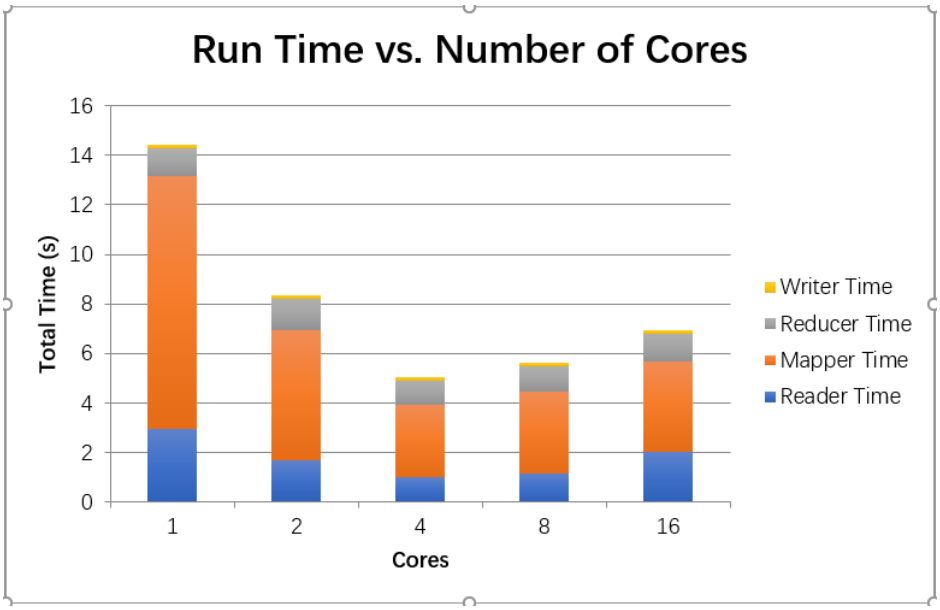
i) OpenMP:

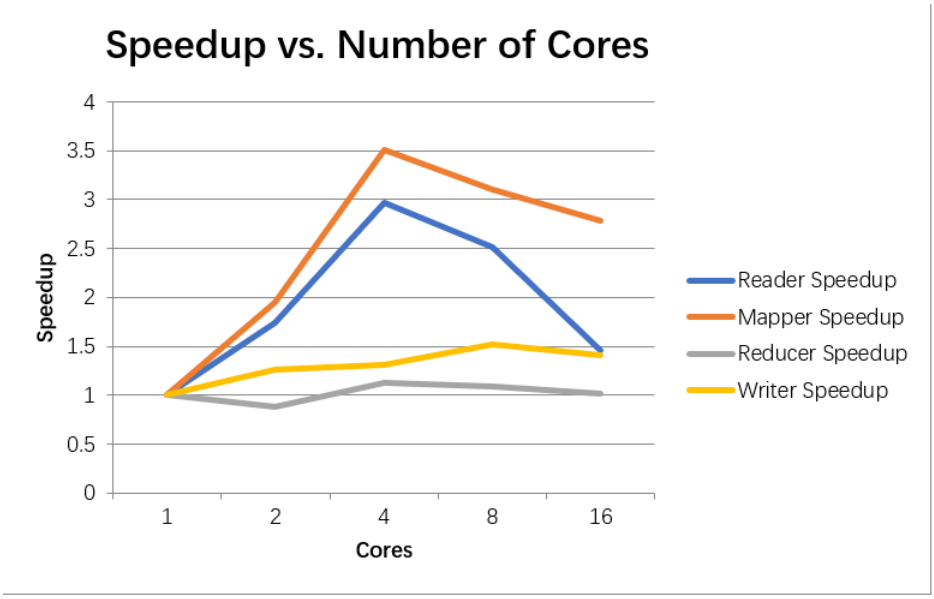
The date in the following table are average run time of 10 runs with same input but different number of cores.

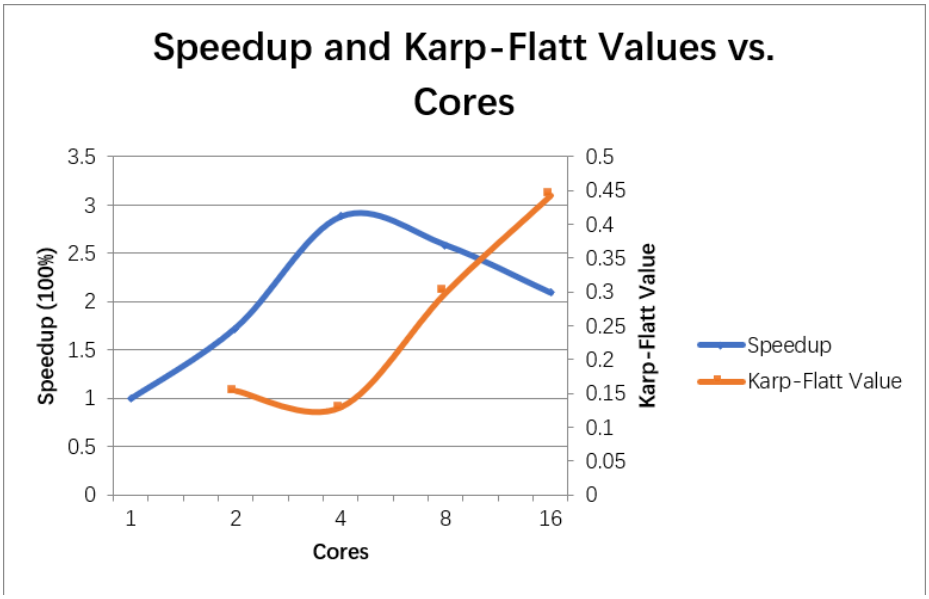
Core(s)	Reader Time	Mapper Time	Reducer Time	Writer Time	Total Time
1	2.936	10.245	1.108	0.156	14.445
2	1.683	5.268	1.254	0.124	8.329
4	0.987	2.915	0.989	0.119	5.01
8	1.168	3.298	1.016	0.103	5.585
16	2.007	3.685	1.098	0.111	6.901

Core(s)	Reader Speedup	Mapper Speedup	Reducer Speedup	Writer speedup	Total Speedup
2	1.744	1.945	0.884	1.258	1.734
4	2.975	3.515	1.12	1.311	2.883
8	2.514	3.106	1.091	1.515	2.586
16	1.463	2.78	1.009	1.405	2.093

Cores	2	4	8	16
Speedup	1.734	2.883	2.586	2.093
Efficiency	0.867	0.721	0.323	0.131
Karp-Flatt	0.153	0.129	0.299	0.443


Run Time vs. Number of Cores
Total Time (s)
16
14
12
10
8
6
4
2
0
1
2
4
8
16
Cores
Writer Time
Reducer Time
Mapper Time
Reader Time


Speedup vs. Number of Cores
Speedup
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
4
8
16
Cores
Reader Speedup
Mapper Speedup
Reducer Speedup
Writer Speedup


Speedup and Karp-Flatt Values vs. Cores
Speedup (100%)
3.5
3
2.5
2
1.5
1
0.5
0
Karp-Flatt Value
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1
2
4
8
16
Cores
Speedup
Karp-Flatt Value

The reader and mapper part have significant speedups when there are 2 and 4 cores. The overhead become larger and parallel algorithm cannot perform best when larger than 4 cores, so the speedup is not that satisfied. Also, the running time of sequential part just fluctuate a bit with larger number of cores but no big difference. Thus, this algorithm works on the parallel part and it has best performance with four cores. Reading is a hardware operation that occurs on one element at a time, and when it becomes the bottleneck, the entire performance will suffer. The total reducer and writer times remain relatively constant for each iteration of the program. The reducer time is going to be quite efficient because it is all local concatenation of hashes, and the writing time by nature will always be fully sequential. In the OpenMP version the write time will gain some overhead as additional cores hit barriers, but overall it is relatively constant. There is a growth in the time required to complete mapping. The mapping logic waits for all the mapping to conclude before it progresses to make sure that nothing is lost in translation over the course of the program. If any mapping is still occurring and a given core moves on, the words that were put in that cores queue after it moved on would be lost. Reduction requires the mapping to be complete. One of the strongest aspects of this algorithm is the way that reduction and writing is intertwined. Because mapping stores values to a core's queue, when the reduction finishes there is a guarantee that that thread's subset of the alphabet is complete. This means that the node with a complete reduction can immediately write those reduced values to memory and move on. As the number of cores increases, the time required for a core to write, and the delay waiting to write will both decrease.

MPI:

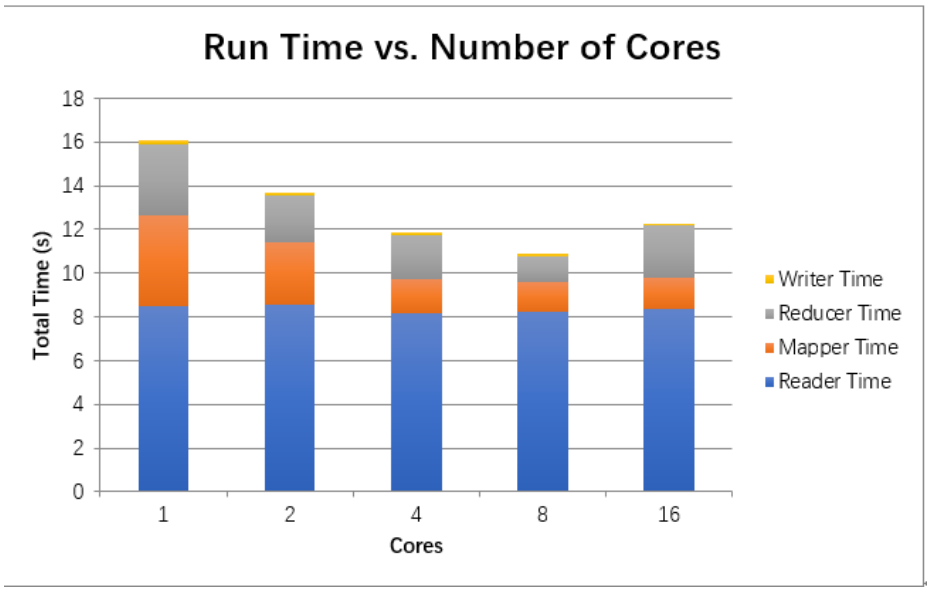
The date in the following table are average run time of 10 runs with same input but different number of cores.

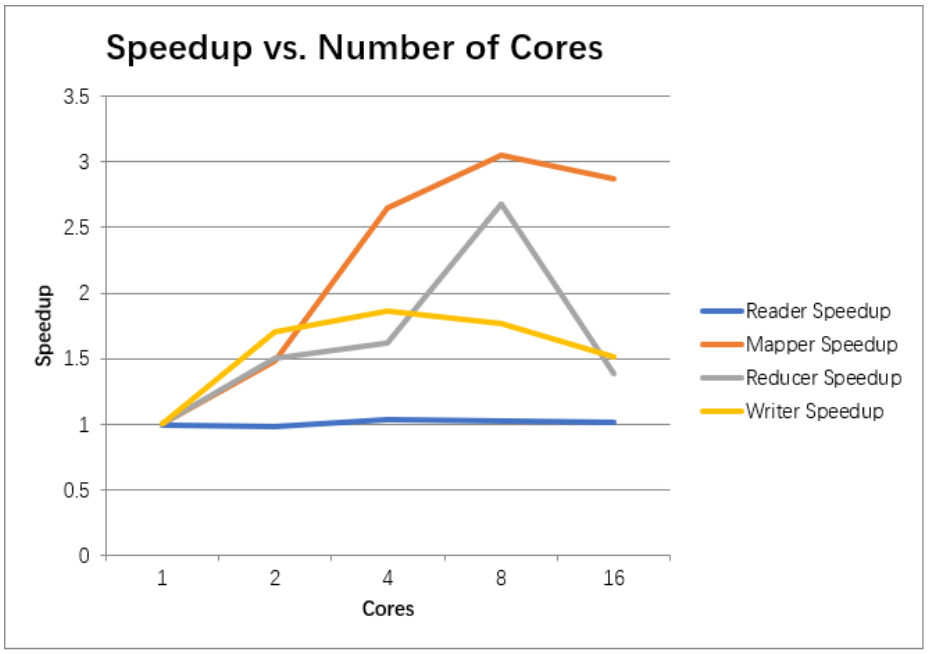
Core(s)	Reader Time	Mapper Time	Reducer Time	Writer Time	Total Time
1	8.462	4.161	3.251	0.175	16.049
2	8.574	2.815	2.165	0.103	13.657
4	8.158	1.571	2.013	0.094	11.836
8	8.205	1.365	1.214	0.099	10.883
16	8.365	1.451	2.341	0.116	12.273

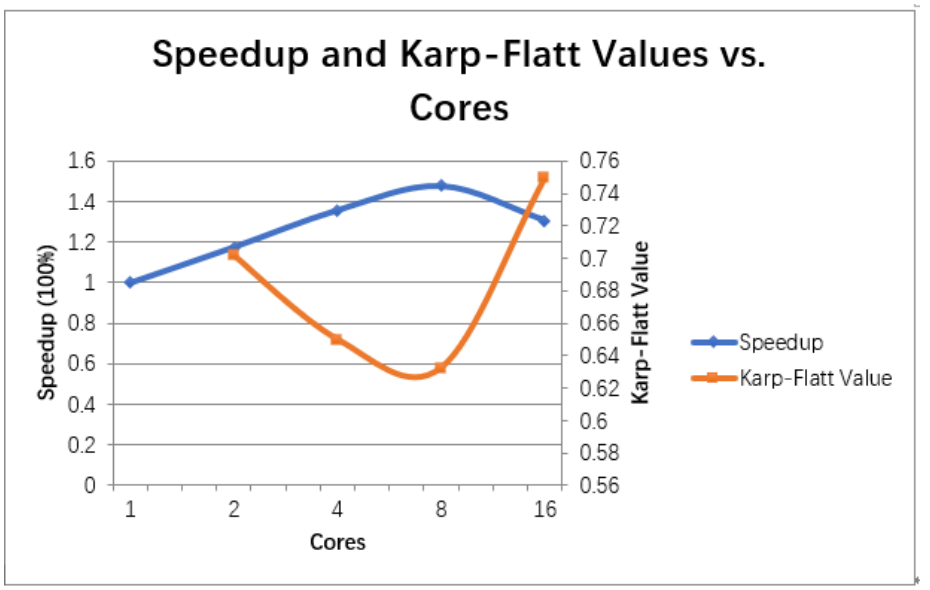
Core(s)	Reader Speedup	Mapper Speedup	Reducer Speedup	Writer speedup	Total Speedup
2	0.987	1.478	1.502	1.700	1.175
4	1.037	2.649	1.615	1.862	1.356
8	1.031	3.048	2.678	1.768	1.475
16	1.011	2.868	1.389	1.509	1.308

Cores	2	4	8	16
Speedup	1.175	1.356	1.475	1.308
Efficiency	0.588	0.339	0.184	0.082
Karp-Flatt	0.702	0.650	0.632	0.749

a


Run Time vs. Number of Cores
Total Time (s)
18
16
14
12
10
8
6
4
2
0
1
2
4
8
16
Cores
Writer Time
Reducer Time
Mapper Time
Reader Time


Speedup vs. Number of Cores
Speedup
3.5
3
2.5
2
1.5
1
0.5
0
1
2
4
8
16
Cores
Reader Speedup
Mapper Speedup
Reducer Speedup
Writer Speedup


Speedup and Karp-Flatt Values vs. Cores
Speedup (100%)
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Karp-Flatt Value
0.76
0.74
0.72
0.7
0.68
0.66
0.64
0.62
0.6
0.58
0.56
1
2
4
8
16
Cores
Speedup
Karp-Flatt Value

In general, MPI has some speedup but it is unworthy since it has too much overhead when compared to OpenMP. When studying the Karp-Flatt values, the general trend is decreasing serial code as the number of cores increases. Despite this trend, the actual speedup not increasing at a constant or even growing rate, instead the rate at which the speedup grows is decreasing. The answer to this problem can be found in the Reducer thread. The reducer thread is consuming an increasing amount of CPU time as the cores grow which makes sense due to the nature of a deepening reduction tree. Even though it is an efficient reduction, doing so causes a significant amount of idle time in threads. The steep growth of idle threads can be blamed for the decreasing speedup rates. The trends seen in the mapper and reader threads are responsible for the speedup that is observed. The steady drop in real time spent in those tasks is because those static operations are being spread evenly among an increasing number of cores. In the MPI version, the reader threads are not competing for access to the hard drive's read head, because these threads are spread across distributed machines instead of all residing locally. The bottleneck in this case is not the memory hardware of the system but is instead the communication hardware. In the MPI model, there is a master thread that is fully responsible for distributing the work (files) that need to be read. This thread manages the master list of files that need to be processed, and because only one MPI operation can be performed at a time, the thread can only service one node at a time. To fix the bottlenecks caused by the reducer threads and take advantage of the performance gain created through the reader and writer threads, a much more intelligent communication model needs to be created.

All the hashes are sent manually, with a starting message, the word, and then the value. This is extremely expensive as a simple object is converted into a complex series of communications that must deal with network traffic and various other induced latencies. To correct for this, the data needs to be sent in a larger package with much fewer requests to avoid paying communication costs multiple times. The easiest way to move to a proper implementation is to define custom MPI data structures or use a library like Boost that has already done it instead of manually transmitting hashes.

IV. Conclusion

OpenMP has higher efficiency and better performance and it is easier to implement the algorithm. MPI version, due to larger overhead of communication between the nodes and trickier algorithm to manipulate the hash table, has a little speedup but is tiny when compared with OpenMP. The reason why MPI performs badly is because that the hash table was implemented locally and did the combination and reduction in order over time. However, the shared memory used globally in the OpenMP so that every thread can access it.