ECE 56300: PROGRAMMING PARALLEL MACHINES

(SPRING 2019)

MAP REDUCE USING PARALLEL PROGRAMMING METHODS

DATE OF SUBMISSION : 26TH APRIL 2019

By: Anirudh Sivakumar and Vaibhav Ramachandran

Table of Contents

Contents

What is Map Reduce?

Map Reduce is a programming model for processing large amounts of data sets using parallel algorithms among many computers usually in a cluster in order to yield better performance. Generally Map Reduce algorithms make use of queues for different parts or stages of the algorithm in order to maintain coherency and atomicity in order to obtain accurate results as these operations are performed over many computers and synchronization becomes important.

Although there are many methods of implementing Map Reduce, these kinds of algorithms are becoming more and more popular due to the fact that the general algorithm can be used to complete a wide variety of tasks in parallel. For example, counting number of unique words in a file, pattern-based searching, distributed sorting, machine learning, etc.

What is the project about?

Our project uses the Map Reduce algorithm to read words from files and output the word count for each unique word that appeared in the files. A path to the file directory is fed into our program and the program spits out result text files which contain the unique words and their frequencies from all files read in the directory supplied to our program.

Our project also makes use of MPI and OpenMP techniques in order to achieve better performance than a sequential version by the use of multi-threaded programs over a cluster of computer cores. The OpenMP model is used in order to get better performance from a single core by the use of multiple threads and the MPI model is used in order to get better performance by the use of many cores thereby having more hardware to complete our tasks faster.

Implementation details

As mentioned before, our project makes use of MPI and OpenMP methods in order to boost performance. Our C++ program takes in the number of processes/cores over which to execute our code on along with the file directory containing the files that the user wants to go over. The results are ‘n’ text files where ‘n’ is the number of processes used. Within these text files would be the results which include the distinct words found in all the files traversed and their frequencies.

Within the program itself, we make use of multiple reader, mapper, sender and receiver threads in order to improve per core performance. Process/core 0 initially takes in the file directory and traverses it to find all the files present in the directory. It then places all the files in a vector data-structure of datatype string and it is from here that all other threads and processes/cores retrieve files to read and process information from. Thread 0 in process 0 is responsible only for the MPI sends and receives of the file paths, i.e., its main purpose is to send and receive filenames to different processes. Each of the other threads in other processes are grouped and have different functions. These groupings of threads are either the receiver threads, mapper threads, sender threads or the receiver threads.

The receiver threads ask for a file, collect words from the file requested, makes sure that the word read is under 45 characters in length, turns the word into all lowercase letters and puts the tuple (<word, word count of 1>) in a queue. Once they have completed this work on a file, they request for another file. This process keeps going until they receive a special file name which suggests that there are no more files to be read. Each thread belonging to the pool of reader threads performs this action. Therefore, if a reader thread pool contains 8 threads, 8 files will be read and processed in parallel. There is a separate queue for each thread where it stores the tuple (<word,1> where 1 is the word count).

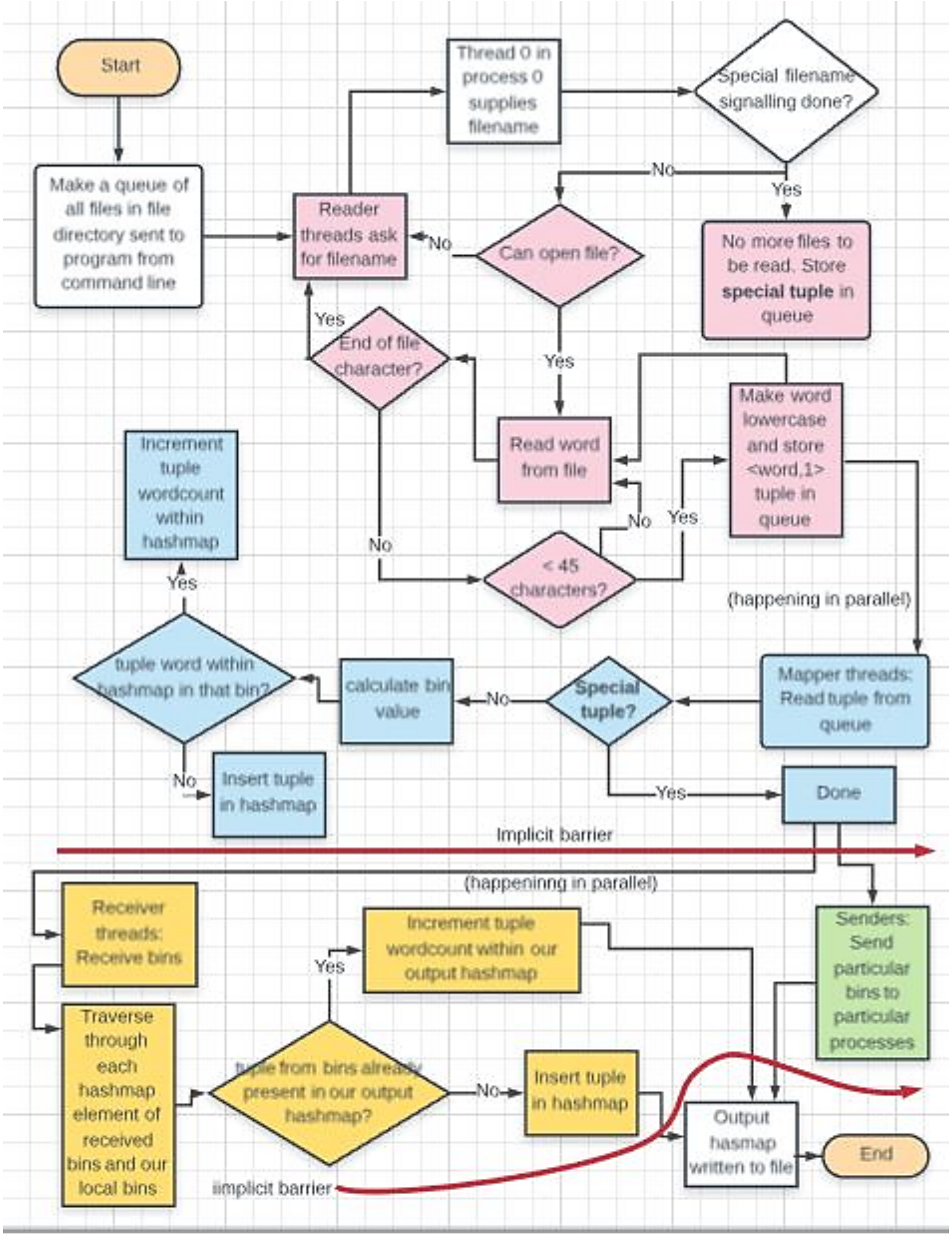
The mapper threads work in parallel with the reader threads. As soon as a queue has words in it, the mapper threads automatically read the words from the queue and create a tuple hash map structure (of <word, wordcount>) where they place these words. New words are inserted into the hash map while words that already exist in the hash map only have the wordcount in the tuple updated. Now, there exist a number of bins each of which have these hash map datatypes where the words are sent to. This is so that once each process finishes mapping all words it read from its corresponding files, it can send certain bins containing these hash maps to certain processes and also receive certain bins from other processes. For this to happen it becomes crucial to make sure that a word falls into the same hash map bin. Therefore the function that oversees this makes use of the first character of the word and the word length in order to calculate which bin the word should fall under. Our bin function was made so that it is fast and accurate and uses the following formula:

$$\text{Bin} = ((\text{int})\ ((\text{first character of word in tuple}) - 97) * (\text{number\_of\_bins}/26) + (\text{tuple\_word\_length}))\ \%\ \text{number\_of\_bins}$$

Once the readers and mappers are finished, there is an implicit barrier after which the senders and receivers begin their work. The sender threads are responsible for sending certain bins (of the bins of hash map that the mappers created) to certain processes. This ensures that each process handles certain number of bins and therefore certain words. The distribution of bins to the processes is done cyclically based on the unique process ID. For example, if there are 4 processes, process 0 would be responsible for all words in bins 0, 4, 8, etc. while process 1 would be responsible for all words in bins 1, 5, 9, etc.

The receiver threads act in parallel with the sender threads. They receive these bins from the sender threads and then integrate these newly received bins with the bins that they already have. Now, all of these bins are traversed and integrated into an output tuple hash map that is then stored into a file.

The following flowchart describes the whole process:


Start
Make a queue of all files in file directory sent to program from command line
Reader threads ask for filename
Thread 0 in process 0 supplies filename
Special filename signalling done?
No
Yes
No more files to be read. Store **special tuple** in queue
Can open file?
No
Yes
Read word from file
End of file character?
Yes
No
< 45 characters?
No
Yes
Make word lowercase and store <word,1> tuple in queue
(happening in parallel)
Mapper threads: Read tuple from queue
Special tuple?
No
Yes
Done
calculate bin value
tuple word within hashmap in that bin?
Yes
No
Increment tuple wordcount within hashmap
Insert tuple in hashmap
Implicit barrier
(happeninng in parallel)
Receiver threads: Receive bins
Senders: Send particular bins to particular processes
Traverse through each hashmap element of received bins and our local bins
tuple from bins already present in our output hashmap?
Yes
No
Increment tuple wordcount within our output hashmap
Insert tuple in hashmap
Output hasmap written to file
End
iimplicit barrier

Scope for improvement

We believe that even though we have been able to extract decent speedups from the program, there is some room for improvement. Some places of improvement include:

- The method of telling readers to stop reading files. Instead of sending a special filename to all readers, maybe just 1 broadcast by any of the threads should give enough information to all threads to stop reading after processing the current file. This way we could probably save on extra communication overheads
- Instead of having 1 thread read through a file, if there was a way to make multiple threads read through a single file. This way, we might be able to read through bigger files faster while also fully utilizing all the threads in a thread pool and not making any threads wait after they are done with their job.
- Rework the algorithm to get better performance with more threads on a single core. The amount of speedup we get from increasing the thread count is not as significant as we would like given the scaling of the number of threads.
- Being able to test on more than 2 nodes. Having run our tests on the scholar machine, we have noticed that jobs take noticeably longer to run on more than 1 node even though the actual running time of the program itself is less than 10 seconds for the number of files used for testing. (For example, running on 2 nodes with 4 cores per node takes ~ 4 hours while running on 1 node with 8 cores per node takes ~ 10 minutes) Therefore, use of a faster testing method and more testing resources would be beneficial in finding out more areas of improvement.

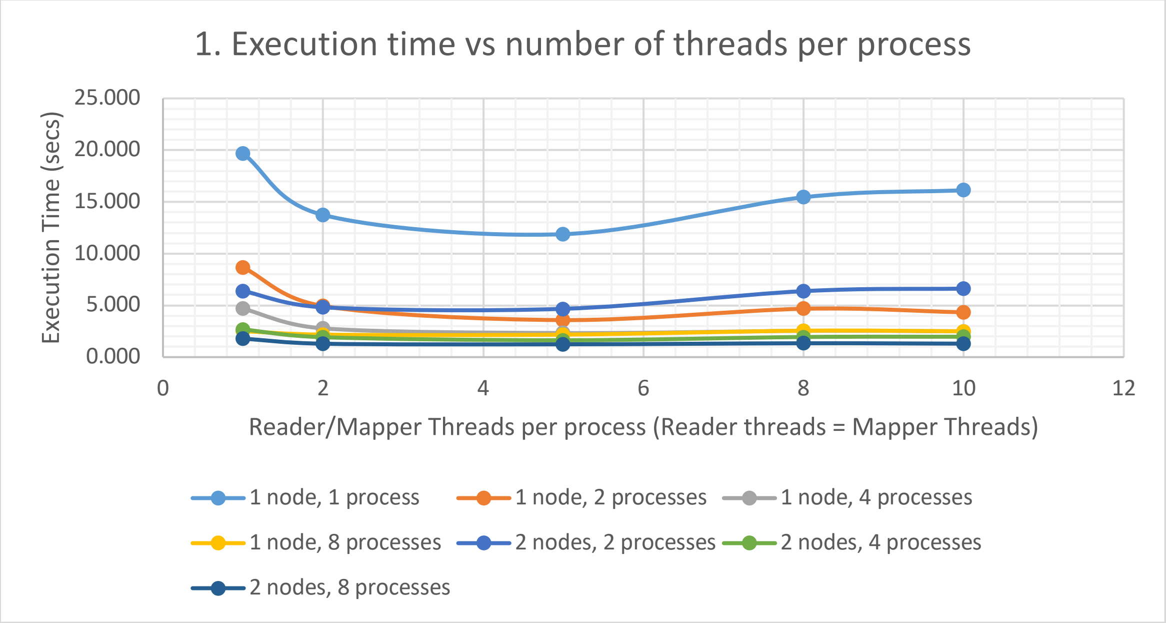
Performance Bottlenecks

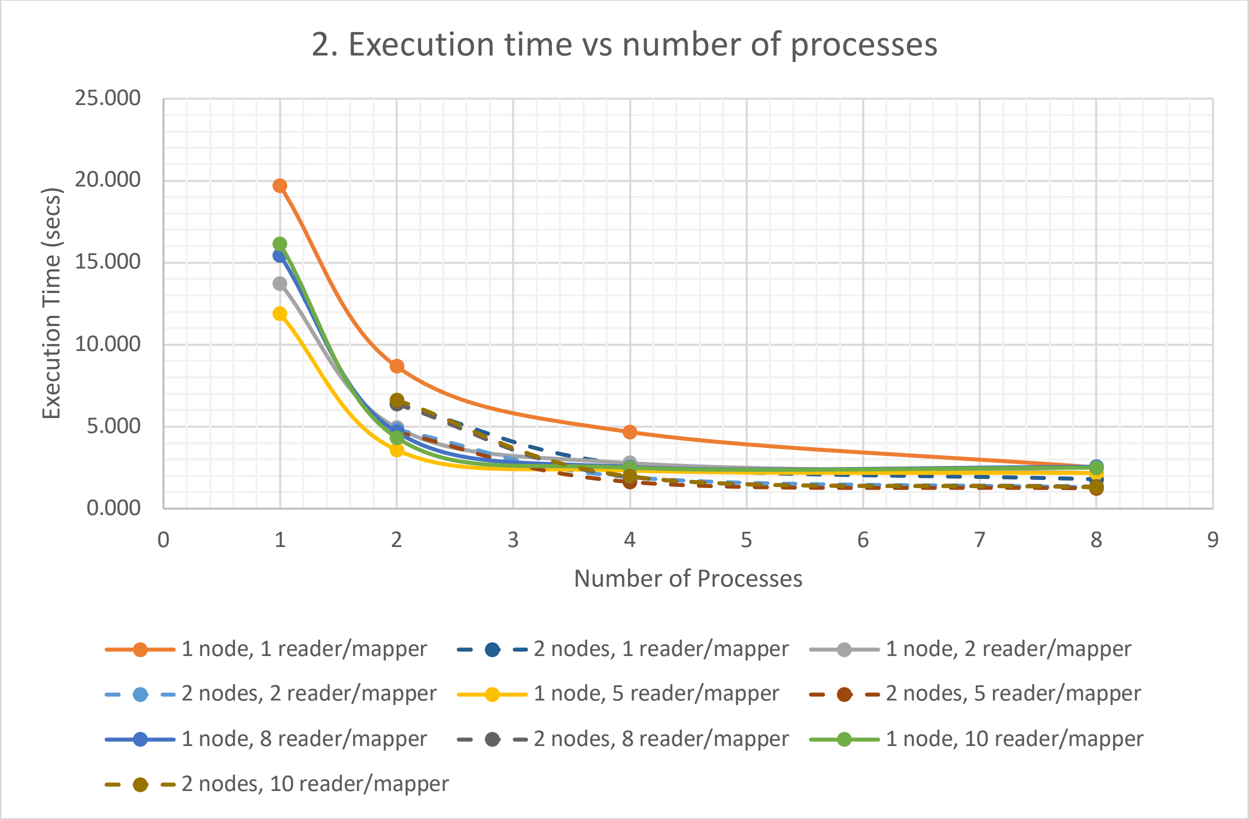
Some of the bottlenecks we observed from running our program on scholar include:

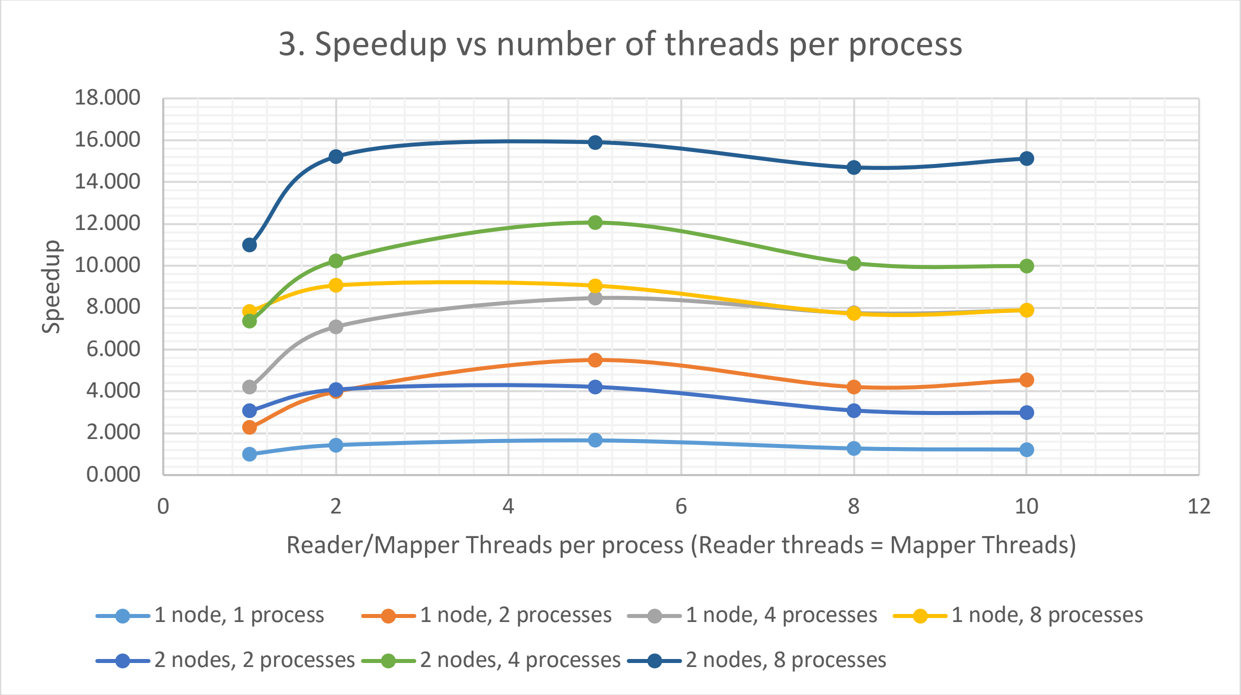
- The mapper threads take the maximum amount of time to execute their portion. This is likely because of a critical section that we placed in the mapper thread function which increases the sequential execution time of the function. The critical section is necessary because if it so happens that multiple threads are simultaneously attempting to write the same word into the map, they could overwrite each other's changes. Eventually this would cause the program to output incorrect data.
- Another bottleneck that we observed that increases the sequential time of the program, is in our receiver thread function. When we receive the bins from each process, we need to atomically update our local bins for the same reason as above. Allowing multiple threads to update the local map concurrently will cause them to overwrite each other's data leading to incorrect results.

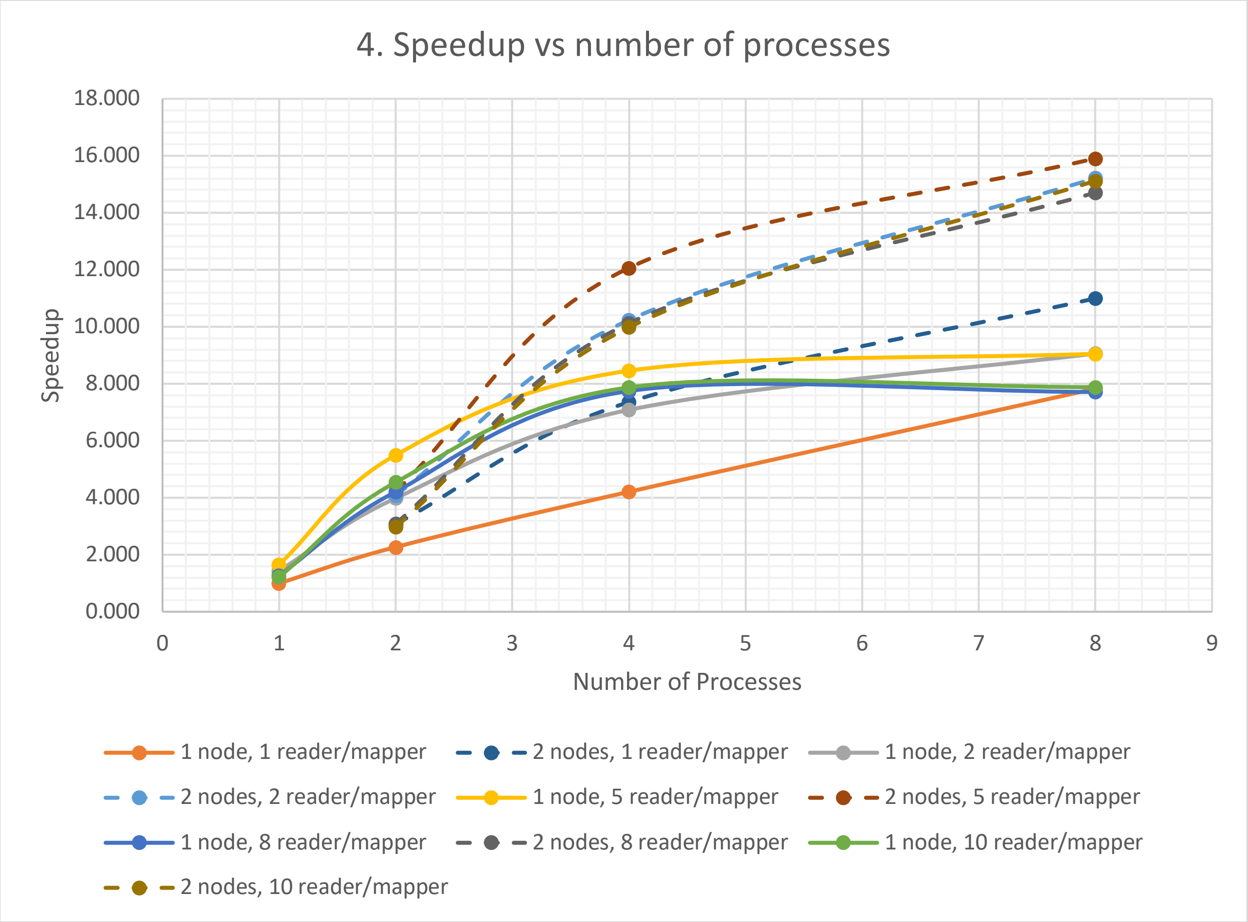
Results and Formulae

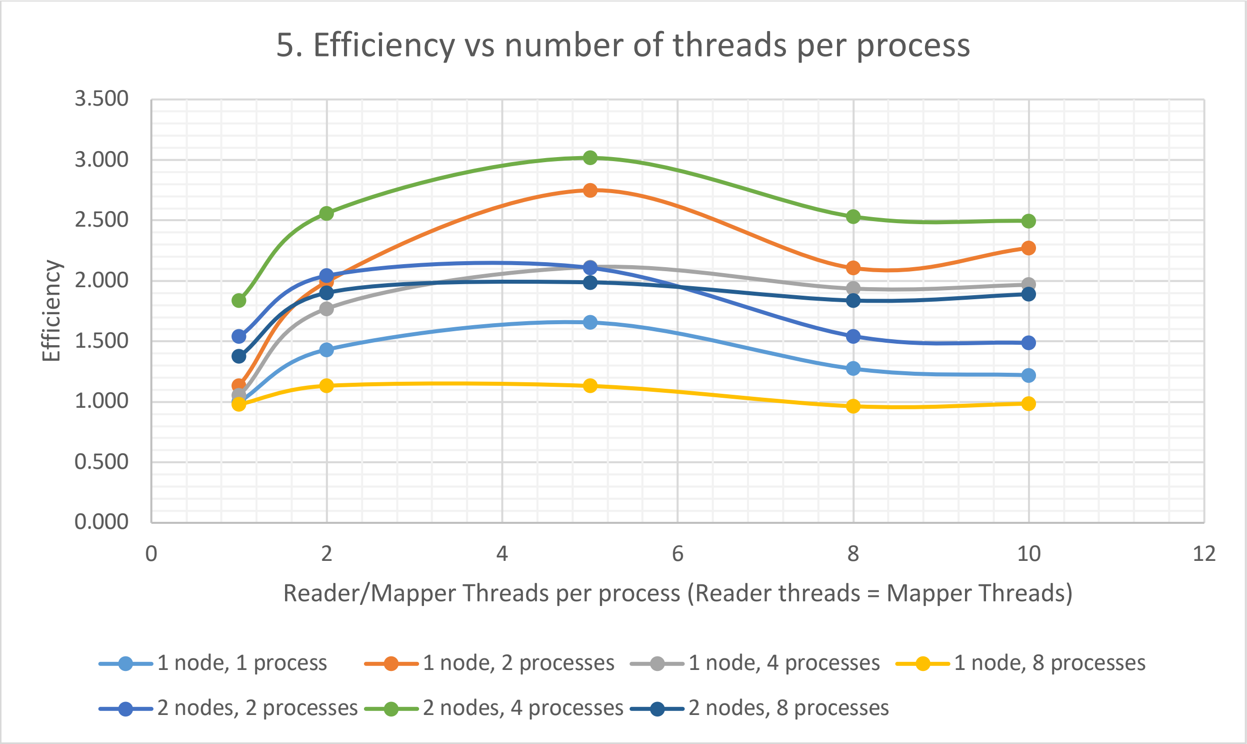
1. Speedup = (Time taken on 1 process, with 1 reader/mapper) / (Parallel time taken)
2. Efficiency = (Speedup) / (Number of Processes)
3. Karp-Flatt = (1/Speedup – 1/Processes) / (1 – 1/Processes)

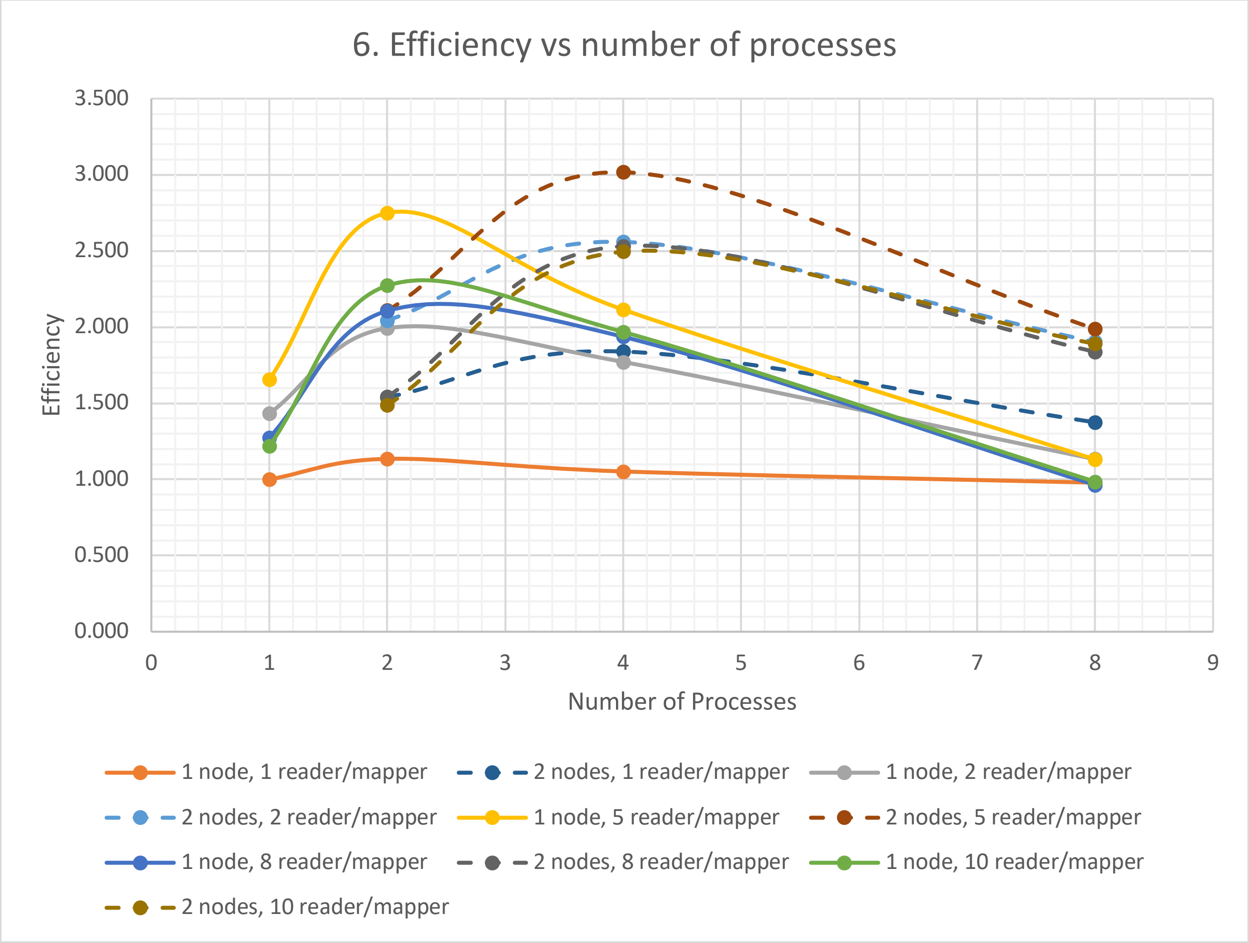




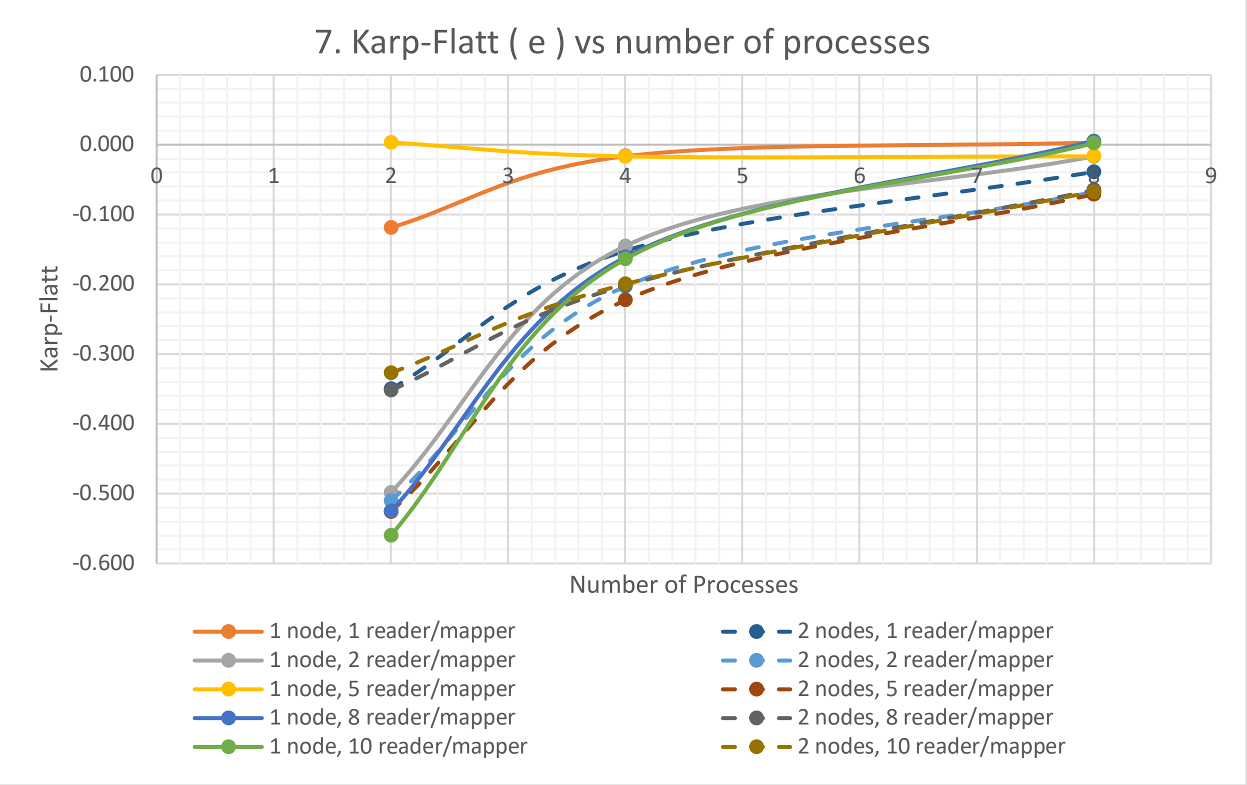

3. Speedup vs number of threads per process
Speedup
18.000
16.000
14.000
12.000
10.000
8.000
6.000
4.000
2.000
0.000
0
2
4
6
8
10
12
Reader/Mapper Threads per process (Reader threads = Mapper Threads)
1 node, 1 process
1 node, 2 processes
1 node, 4 processes
1 node, 8 processes
2 nodes, 2 processes
2 nodes, 4 processes
2 nodes, 8 processes


4. Speedup vs number of processes
Speedup
18.000
16.000
14.000
12.000
10.000
8.000
6.000
4.000
2.000
0.000
0
1
2
3
4
5
6
7
8
9
Number of Processes
1 node, 1 reader/mapper
2 nodes, 1 reader/mapper
1 node, 2 reader/mapper
2 nodes, 2 reader/mapper
1 node, 5 reader/mapper
2 nodes, 5 reader/mapper
1 node, 8 reader/mapper
2 nodes, 8 reader/mapper
1 node, 10 reader/mapper
2 nodes, 10 reader/mapper


5. Efficiency vs number of threads per process
Efficiency
3.500
3.000
2.500
2.000
1.500
1.000
0.500
0.000
0
2
4
6
8
10
12
Reader/Mapper Threads per process (Reader threads = Mapper Threads)
1 node, 1 process
1 node, 2 processes
1 node, 4 processes
1 node, 8 processes
2 nodes, 2 processes
2 nodes, 4 processes
2 nodes, 8 processes


6. Efficiency vs number of processes
Efficiency
3.500
3.000
2.500
2.000
1.500
1.000
0.500
0.000
0
1
2
3
4
5
6
7
8
9
Number of Processes
1 node, 1 reader/mapper
2 nodes, 1 reader/mapper
1 node, 2 reader/mapper
2 nodes, 2 reader/mapper
1 node, 5 reader/mapper
2 nodes, 5 reader/mapper
1 node, 8 reader/mapper
2 nodes, 8 reader/mapper
1 node, 10 reader/mapper
2 nodes, 10 reader/mapper

As can be seen from the above plots 1, 3 and 5, performance (execution time), speedup and efficiency improve as the number of threads per process increases. However, this trend is only valid until a certain limit. Each node on the scholar cluster consists of 20 threads and when the total number of threads in all processes exceeds the threads available, we see a performance drop. This is likely because of threads getting swapped out and those overheads are causing a drop off in performance, which in some cases (1 or 2 processes on 1 or 2 nodes) can be quite significant. As expected from plots 2, 4 and 6, the performance improves exponentially as the number of processes are increased exponentially. This is because of the increased hardware executing the program. Another thing to be noted is that the performance improves even more if the number of nodes used also increases. This is seen from 2, 4 and 6 for the runs with double the number of nodes, keeping the number of threads per process constant. For the same number of processes and threads per process, using double the number of nodes provides a significant speedup and is more efficient as a result.



The Karp-Flatt metric allows us to gauge the amount of sequential work and communication being done in the program. From this plot it can be seen that the Karp-Flatt metric is increasing for all our runs. This effectively means that the amount of sequential work is increasing as is the amount of communication as our processes increase. This makes sense since the increase in processes means an increase in communication. Allocating more threads per process causes the Karp-Flatt value to significantly increase with increase in processors. This is because of the performance bottlenecks mentioned in the previous section. More time is spent in these sequential sections and as such performance, efficiency and speedup suffer.

1. Average Overall Times (seconds)

Number of Threads ➡		1	2	5	8	10
Number of Nodes ⬇	Number of Processes ⬇					
1	1	19.677	13.737	11.880	15.446	16.129
1	2	8.669	4.937	3.579	4.671	4.330
1	4	4.675	2.777	2.326	2.541	2.500
1	8	2.514	2.172	2.174	2.552	2.497
2	2	6.391	4.816	4.667	6.375	6.620
2	4	2.674	1.922	1.631	1.943	1.971
2	8	1.789	1.293	1.237	1.338	1.301

2. Average Reader Times (seconds)

Number of Threads ➡		1	2	5	8	10
Number of Nodes ⬇	Number of Processes ⬇					
1	1	17.404	11.374	9.619	13.146	13.954
1	2	8.478	4.722	3.286	4.275	4.033
1	4	4.219	2.358	1.535	2.005	2.011
1	8	2.011	1.604	1.632	1.698	1.694
2	2	6.199	4.458	4.482	6.202	6.180
2	4	2.402	1.667	1.391	1.710	1.689
2	8	1.426	0.935	0.815	0.912	0.864

3. Average Mapper Times (seconds)

Number of Threads ➡		1	2	5	8	10
Number of Nodes ⬇	Number of Processes ⬇					
1	1	17.411	11.376	9.622	13.151	13.958
1	2	8.484	4.730	3.287	4.282	4.036
1	4	4.225	2.363	1.899	2.011	2.014
1	8	2.015	1.610	1.637	1.703	1.700
2	2	6.202	4.463	4.487	6.205	6.184
2	4	2.405	1.671	1.395	1.714	1.693
2	8	1.431	0.939	0.820	0.916	0.869

4. Average Sender Times (seconds)

Number of Threads ➡		1	2	5	8	10
Number of Nodes ⬇	Number of Processes ⬇					
1	1	2.173	2.242	2.176	2.222	2.067
1	2	0.106	0.068	0.066	0.099	0.083
1	4	0.350	0.322	0.332	0.364	0.440
1	8	0.427	0.468	0.476	0.699	0.603
2	2	0.095	0.057	0.075	0.100	0.095
2	4	0.186	0.168	0.168	0.176	0.182
2	8	0.287	0.273	0.310	0.342	0.315

5. Average Receiver Times (seconds)

Number of Threads ➡		1	2	5	8	10
Number of Nodes ⬇	Number of Processes ⬇					
1	1	2.236	2.327	2.230	2.277	2.145
1	2	0.171	0.160	0.240	0.174	0.136
1	4	0.439	0.359	0.418	0.458	0.475
1	8	0.452	0.498	0.508	0.731	0.744
2	2	0.167	0.227	0.284	0.147	0.335
2	4	0.208	0.233	0.212	0.206	0.270
2	8	0.346	0.327	0.396	0.399	0.404

All runs were done with 128 files read.

Conclusion

The amount of work done is directly proportional to the number of words that are read. Thus, the iso-efficiency function is of order n where n is the number of words or work. Since our program has data level parallelism, increasing the number of processes should provide an improvement in performance, which is in line with our results. Increase in speedup when thread count increased was satisfactory. Since there is no definitive measure of performance improvement vs thread count, we couldn’t tell if our program scaled well or not, Nevertheless, we did get increased speedup up to an extent by increasing the number of threads especially for larger amounts of work. This could be because with more parallel work, the overheads of creating threads was hidden/handled well. However, the program did perform extremely well when the number of processes was scaled, giving exponential speedups. Further speedups were achieved by increasing the number of nodes that the processes were allocated on.