MapReduce

By Austin Ketterer

<u>OpenMP Strategy</u>

The strategy with my OpenMP implementation was to split up the files evenly between the number of threads, then have each thread read through the file and when it finds a word, it hashes it and stores it into a shared map. In order to ensure consistency in the map there are 1000 locks distributed over the 13337 map entries. This means that about every 13 values has a lock associated with that group. The number 13337 was chosen because after looking online I found that the average book would have around 10000 unique words. It seemed that having a hash table about 1.3 times bigger was good, as well as using a prime value, so I went with 13337. Any conflicts were dealt with by making each of the entries a linked list that could queue on more words. The true number turned out to be around 50,000 different words from all the books. I believe this is due to it covering books from many different time periods, where useage of the english language varies widely.

Ideally how I planned it out in my head, there would be little conflict with such a big map. The hash seemed to evenly distribute across the data set. The hash that I used was multiplying each character of the string by 31^{position in string} then doing modulus 13337. This should have meant that the chance of one of the processes waiting on another would be really low. The one issue with that is that 1 letter words were very common. As they would all get a hash value close to each other they likely caused some slowdown in each other due to sharing the same lock. If I wanted to improve this in a future version I would make locks for each of the individual letters since they are popular or find a better hash. Or another thing that comes to mind now after the fact, would be using 31^{position in the string + 1}. This would separate the letters enough to give them separate locks. This program does work well for books where there are many different words, but this is not the case for all data types that someone may want to MapReduce.

I had based my program off of books which are fairly random, but some datasets see a lot of repetition. For instance when I was first testing, I used 16 copies of the same file. This meant that my program went super slow, even slower than one thread. All that was happening was that all the threads would be waiting for the same lock because they were going through the same file at the same pace. Even when I tried to randomize and increase the size of the dataset, I still had this issue somewhat. To make a large amount of data to test on I repeatedly copy and pasted a few of the books and whenever two threads would be reading through the same book at the same time, they would perform slower than the rest of the threads. This is definitely a big design flaw if the data set features a lot of similar data.

Even though the speedup isn't great at two threads, it at least consistently speeds up as the number of threads goes up. My test used 27MB worth of data that was queued up 100 times on each node for a total of around 2.7GB of text.

Data for OpenMP Implementation

Number of Threads	Time (seconds)	Speedup	Efficiency	Karp-Flatt
1	130.6	1	1	
2	102.48	1.274395004	0.637197502	0.5693721286
4	58.66	2.226389362	0.5565973406	0.2655436447
8	30.82	4.237508112	0.529688514	0.1268431415
16	16.24	8.041871921	0.5026169951	0.06597243492

One thing which slows the program down is that one thread will always finish a bit later than the other ones. I tried to split the text evenly but this doesn't really account for the fact that different threads may go through the data at different speeds. For instance in my 16 thread trial, Thread 5 finishes at 14.81 seconds whereas Thread 12 finishes at 12.6 seconds. One way to solve this issue would be to have everyone pulling off of a shared work queue but I do not think that the overhead of doing that is worth it.

I attempted to do this in fact to fix to my program. I implemented this by giving each thread its own queue of words it could go through, and the ability to pull from other's queues when it was empty. This however provided a major slowdown for the program. From a speedup of 8.04 with 16 threads to around 3. I believe this is due to too much data being on the heap. I even tried to use tasks to get the readers and mappers executing simultaneously but this still involved a large slowdown for the program. This did mean that all of the threads finished at almost exactly the same time but it slowed down the process too much to be worth it.

Even though the speed increases aren't large, it at least has a Karp-Flatt metric that is going down. This means that this program is scaling better as it gets bigger. The majority of the program is purely parallel. The only serial part is the printing at the end which took only an additional 0.02 seconds at the end of the program. I did write some code to parallelize this, but at such a miniscule portion of the time anyway I did not think it was worth it to do it when it would mean splitting up the output in to separate files. This should mean that the program will scale very well as the vast majority of time is spent in the parallel region, and threads waiting on the same lock is unlikely.

OMP/MPI Hybrid Implementation

I hate MPI. I spent around 10 hours trying to figure out what was causing https://i.imgur.com/WFjG4MK.png but the only thing that worked for me to fix it was completely not using any functions at all (Including any functions even ones that I did not use would cause MPI_Init() to segfault). So what I had to do is completely rebuild the algorithm without any functions. This took a lot of time .

How I decided to approach the MPI implementation was very similar to how I did my OpenMP implementation. Instead of sending all the words around which would involve a lot of communication, I would just finish all the mapping first and then send the data. This would mean on words like "a" I saved the work of around 2.26 million sends.

Data For OMP/MPI Hybrid Implementation (6.69 GB)

Number of Nodes	Time (seconds)	Speedup	Efficiency	Karp-Flatt
1	41.17	1	1	
2	21.45	1.919347319	0.9596736597	0.042020889
4	11.12	3.702338129	0.9255845324	0.02679944944
8	6.01	6.850249584	0.856281198	0.02397723724

The redistributing of the data at the end was done by setting a number of receiver and sender threads running at the same time. The sender threads would distribute the mapped data to different nodes based on the hash value of the string. I chose to use 10 receivers and 6 senders. I will admit this was somewhat arbitrary. The amount of time that the redistribution took on two nodes was just .14 seconds or 5% of the total runtime. I think I would have to use a lot of data and occupy a larger number of nodes in order to fine tune this number well, and it is already a small amount that could vary just based on the random factors in the execution. On eight nodes there is a lot more communication overhead as it takes .7 seconds or 11.5% of the runtime. From looking at that Karp-Flatt numbers 8 nodes was still more efficient than 4 nodes, but I feel that the knee may be at 16 nodes.The percent of time communication takes will only grow (n^2) as the number of nodes gets larger, making it a poor choice for a supercomputer that may have hundreds of nodes. This could be fixed by MapReducing in 16 node groups then reducing all of those files.

Overall the speedup was very close to the ideal speedup when increasing the number of nodes. Other than the communication at the end, no other part of the program has to communicate.

Final Notes

I really enjoyed this project other than having to code everything in the main for MPI. It was a lot of fun trying out different solutions and analyzing how it affected the runtimes. I'm pretty happy about how both of the programs turned out, although I'm still wondering a bit why the speedup is so bad for the OMP version on two nodes. If you have any feedback on my program I would love to hear it. Thanks for the great semester!

Sincerely,

Austin Ketterer