Assignment № 2

Arun K Rajagopalan, Texas A&M University 10/15/2015

Problem

Given an input graph containing an adjacency list of connected nodes, output a final sorted file containing the counts along with node name.

Analysis

The algorithm can be split into 4 stages -

1. Graph split and sort
2. Graph merge
3. Map split and sort
4. Map merge

Each of these stages have been designed to operate independently of the previous stage. By treating each stage as a black-box, I was able to efficiently debug and ensure correctness.

Graph Split and Sort

Algorithm

I present an overview of the Algorithm at a high level.

- Since we are given a total of 500 MB to work with, I had to efficiently manage the Read and Write buffers. I chose to keep a constant fixed 32MB Read buffer and left the rest of my memory for the write buffer. This enabled me to perform maximum compaction before writing to disk.
- I wrote over the HeaderGraph struct while reading the edges. This was done so that I end up with a chunk of data consisting of just the edges. If the HeaderGraph struct + edges did not fit inside the Read buffer, I perform disk seeking to move back and read again.
- Once I have all the edges in my buffer, I write them into my Write buffer as a HashCount struct giving each edge a count of 1. (This is essentially the ‘map’ phase of map reduce).
- The next step involves sorting the edges on the hash key. I used std::sort for this.
- Once sorting is done, we can now compact similar structs together, incrementing the count.

- Finally, once sorting and compacting are both done, we write a new file to disk.
- This process is repeated until we have read the entire input graph.

Optimizations

There are multiple avenues of optimization that I looked into.

- Using ReadFileEx and WriteFileEx. This allowed me to achieve optimal read and write times.
- Treating my write buffer as a 4-bucket buffer. Using the top two bits of the hash, I wrote into the appropriate bucket. This allowed me to call std::sort in 4 separate threads at the same time, almost halving the overall time of this step on my machine.
- Use of two semaphores to act as a lock. Each semaphore was initialized to a value of 4. This allowed the threads to wait while the master processed and inserted data into the write buffer.

Areas of future improvement

While I was able to achieve good speedup with the above mentioned techniques, these we some of the areas I would have also liked to improve.

- Do not disregard the data that is read in if it doesn't fit in the Read buffer.
- Use of a Incremental Hash table as opposed to the sorting I did. This will allow me to improve my write size so that the subsequent merge phase has a smaller depth.
- I currently use ReadFileEx and WriteFileEx in a synchronous manner, which sort of defeats the purpose. A more asynchronous approach would have been preferred.
- ReadFileEx and WriteFileEx should disable File buffering.

Graph merge

Algorithm

I present an overview of the Algorithm at a high level.

- The output files from the previous stage are stored in a queue. I pick two elements at a time, merge and put the result back into the queue.
- This process is repeated until there is exactly one item in the queue - the final merged file.
- Similar to the above step, I used a buffer of size 36 MB to serve as my Read Buffer. Since we need to read from two separate files at the same time, the overall read buffer size is 72 MB. The remaining is left to be used for the Write buffer.

- To simplify reasoning, I created wrapper functions (has_next(), current(), next(), put() etc). These functions greatly simplified the task as I could think of each file + buffer in terms of arrays stored in memory.
  - Every call to has_next() checked two things :- 1) whether the read buffer has elements to process; 2) whether the file handle still has elements to provide.
  - next() provided me the next element to merge, and loaded data back into the buffer if we reached the end of our buffer.
  - put() tried to write to the write buffer - if we reached the end of the write buffer, we write to disk, reset the pointer to the start and insert the element.
- Finally, once we are done reading the file, I call delete on it.

Optimizations

There are multiple avenues of optimization that I looked into.

- Since each stage of the merge is decoupled from the other stages, I was able to split the stages among 2 threads. One half of the files were given to thread 1, the second second to thread 2. Both these threads were then started. Upon returning, we end up with 2 merged files. The master then picks up and processed the files in a 2-way merge.
- Each merge takes 3 (two for read and 1 for write), that gives us a total of 6 open file handles. Since we only have 5 file handles to work with, the two threads opened their write handles using a semaphore and closed them immediately after. This gave us a total of 2 + 2 read handles and 1 write handle.
- Avoiding opening and closing file handles often.

Areas of future improvement

While I was able to achieve good speedup with the above mentioned techniques, these we some of the areas I would have also liked to improve.

- A producer-consumer queue interlocked with the first stage would have provided me with better results.

Map Split and Sort

Algorithm

I present an overview of the Algorithm at a high level.

- Since character buffers are of variable length, I cannot use a normal sort. The data structures I use to perform this are
  - A 36 MB Read Buffer

  - A 9 MB *sizeof(char*) String Pointer buffer. This is needed since the size of pointer changes between 64 bit and 32 bit. This way I avoid taking up extra memory when not needed.
  - A Copy Buffer and
  - a Write buffer of the same size

- I write data into my copy buffer from the read buffer after merging the hash with the character buffer.
- Each time, I store the start address of my string into the String Pointer Buffer.
- When I finally run out of copy buffer size, I then sort the String Pointer buffer with a custom sort comparison that checks the value it is pointing to.
- Once sorted, I then go pick up my structs and write them to the write buffer in sorted order.
- Finally, I ouput the write buffer to disk.

Optimizations

- Initially, I performed this step using std::vector and std::string. The memory consumption shot up very quickly. Switching to the above algorithm really helped performance

Areas of future improvement

I do not see an easy way to parallelize this step.

Map merge

Algorithm

I present an overview of the Algorithm at a high level.

- The approach I follow is analogous to the one for graph merge.

Optimizations

- The very last step of the merge can be combined with writing the txt output.
- I benchmarked sprintf to be the slow on a high performance machine. I used an open source printf module originally written for micro-controllers that had much better performance. I took the source from sparetimelabs.com/printfrevisited and have attributed the author in code.
- A similar parallel merge to the graph merge. I encountered the same problem with open file handles and used the same semaphore to synchronize.

Areas of future improvement

- A more parallel solution or a 4 way merge at each step.

Non-trivial and interesting points

- All file handles should be closed before delete.
- After parallelizing, some of the file handles didn't close as expected. When I tried to delete the file, the operation fails.
- I was able to fix this by localizing the handle open and close to as small a region as possible, to account for unexpected thread exits.

Statistics

Following is the output when run on my personal Macbook Pro (Intel Core i5-4278@2.6GHz, 8GB RAM, SSD) running Windows 10.

```
Starting split phase
Total IO: read - 14.73 GB; write - 5.59 GB
Took 111 seconds
Generated 47 files
Ending split phase

Starting merge phase
Total IO: read - 17.63 GB; write - 13.09 GB
Took 46 seconds
Ending merge phase

Staring map split phase
Total unique nodes - 86533762
Total IO: read - 3.35 GB; write - 1.96 GB
Took 18 seconds
Ending map split phase

Starting map merge phase
Total IO: read - 7.30 GB; write - 7.83 GB
Took 36 seconds
Ending map merge phase

Overall stats - RunTime: 212 seconds;
Total read 43.02 GB; Total write 28.46 GB
```

```
Peak working set : 499.24 MB
DONE!
```

Top 10 links

```
1 microsoft.com 2947630
2 google.com 2209834
3 yahoo.com 1996682
4 adobe.com 1287417
5 blogspot.com 1203819
6 wikipedia.org 1031992
7 geocities.com 932753
8 w3.org 932359
9 msn.com 803930
10 amazon.com 744919
```

ATTRIBUTION

I took help from the following sites to help with my assignment

1. stackoverflow.com
2. MSDN
3. sparetimelabs.com/printfrevisited