1. Abstract

This project realizes building the inverted file from multi-document using parallel programming with Message Passing Interface (MPI).

The whole idea of this program can be roughly divided into 4 procedures:

1. The master scans the files within the given directory, stores the filenames in a vector object, sorts the files in decreasing sizes, and passes filenames to workers one by one.
2. In receiving a filename from the master, the worker will read the file, create a map object to hold the file, invert the file, and store the inverted file in another map object. After that, the worker will serialize the inverted file and send it back to master.
3. In receiving serialized inverted file from any worker node, the master will deserialize it, and merge it into a map object. Keys of the map are strings of words, and values of the map are vectors of integers recording positions of the word in the original book set.
4. Once all the inverted files are merged in the map object, the master will serialize the map object, and write it into a file with the given filename in command line.
5. MASTER: scan files and prepare to delegate jobs

In the provided sample code, the master scan files within the given directory, and stores the filenames in a vector.

1. /book/psalms.txt    size = 241077
2. /book/jeremiah.txt  size = 232973
3. /book/ezekiel.txt   size = 213991
4. /book/genesis.txt   size = 207327
5. /book/isaiah.txt    size = 202647
6. /book/numbers.txt   size = 184240
7. /book/exodus.txt    size = 177563
8. /book/deut.txt      size = 152778
9. /book/2chron.txt    size = 144554
10. /book/luke.txt      size = 142056
11. ......

Figure 1 First 10 files when sorted in decreasing file size

Since the files are very different in size, time in reading a file and parsing the file into an inverted file can be very different. If the files are randomly distributed to the workers, it could happen that all but one worker have completed their work, and the master has to wait for the last worker for a long time, who is coping with a large file. If this case happens, the total program execution time will be long.

To balance the workload among workers, and avoid master’s long waiting time for the last worker, we can delegate the files to workers from the largest to smallest. To this end, we need to sort the files in decreasing sizes. Bubble sort algorithm is used in the sorting since the amount of files is not large, and it won’t take long. Figure 1 shows the first 10 files with their sizes in decreasing size order.

Besides, the master has to get the sequential number of a file in alphabetical sorted order. This information will be used in writing the inverted file to a text file in alpha-numeric order of the indexed terms. Figure 2 shows the first 10 files with their sizes in alphabetical order.

1. /book/1chron.txt    size = 117249
2. /book/1corinth.txt  size = 52965
3. /book/1john.txt     size = 13461
4. /book/1kings.txt    size = 132436
5. /book/1peter.txt    size = 14166
6. /book/1samuel.txt   size = 134853
7. /book/1thess.txt    size = 10173
8. /book/1timothy.txt  size = 13337
9. /book/2chron.txt    size = 144554
10. /book/2corinth.txt  size = 34043
11. ......

Figure 2 First 10 files when sorted alphabetically

Now master is ready to delegate jobs to each worker. By using the message passing code of the boilerplate code, the master sends the filename of a file to each worker waiting for projects. Note that the master has to keep track of which worker is working on which file.

1. WORKER: file reading and parsing

After initialization, the workers will wait for filenames from the master. Once a worker gets a filename through MPI, it will get started to read the file, parse the file line by line, and store the whole book in a *book\_type* map, where *book\_type* is defined as below.

typedef std::map<int, std::string> book\_type;

Key of the *book\_type* is an integer converted from chapter and verse in the following way:

chapterVerseIndex = chapter \* VERSES\_PER\_CHAPTER + verse;

In the code VERSES\_PER\_CHAPTER is set to 10000, which is enough for the books provided in /data/scottgs/book. If verse count within certain chapter is larger than VERSES\_PER\_CHAPTER, error will occur in the runtime, and we have to increase this value.

Now the book is parsed into a *book\_type* object. Next, the *book\_type* object will be converted into another map object called word2index:

std::map<std::string, std::vector<int>> word2index;

where key in string type is the word, the value in vector of integers are chapterVerseIndex. Note that for each word, its chapterVerseIndex will be in a sorted order, since the book is originally parsed from top to bottom, and the items in *book\_type* are stored in a sorted order (attribute of std::map).

Before sending back the map object word2index to master, we need to serialize it into a long string, with added delimiters between different words and different chapterVerseIndex of one word. Since the serialized long string is much shorter relative to std::string.max\_size(), it is safe from string overflow problem.

Strings generated from different books/ files could be in different lengths, so how to coordinate between the workers and the master sizes of the messages. Actually, there are multiple ways.

In the first scheme, the worker could first tell the master how large the string it is going to transmit will be, letting the master provide an enough big container to hold the incoming string message. After the master acknowledges that the container is prepared, the worker starts to transmit the long string to the master.

In the second scheme, the worker and master both have a container in equal size. At the worker side, if the string is larger than the container size, it will be chopped into several segments and transmitted for several times. At the master side, received segments will be concatenated together and the string will be recovered.

My code uses the second scheme, and the container size WORKER\_TO\_MASTER is set to 2001. With the last byte ‘\0’, actual payload is WORKER\_TO\_MASTER – 1 = 2000 bytes.

1. MASTER: merge received books and output to file

Once the master receives one book from a worker, it will send another filename back to the worker, if there is any left. Also, the master needs to deserialize the long string from the worker and merge the received books together into one map object:

std::map<std::string, std::vector<**int**>>& books

Key of the map object books is word, and value is a vector of integers, which are converted from file #, chapter #, and verse # in the following way:

BookChapVerseIdx = fileID \* VERSES\_PER\_CHAPTER \* CHAPTERS\_PER\_BOOK + chapterVerseIndex

In my code, CHAPTERS\_PER\_BOOK is set to 1000, so a book can hold up to 1000 chapters. If a certain book has more than 1000 chapters, error will occur in the runtime, and we need to increase the value of CHAPTERS\_PER\_BOOK.

Note that long strings of books sent to master are not in the alphabetical order, so we need to sort the vector of BookChapVerseIdx before serializing it and writing it into a file. With the parsed fileID we could find the right short filename (without path to it) and write it out to file in the required format.

1. Timing statistics
   1. Total file read time

Figure 3 shows the trend of total file read time with the number of workers increasing. Specifically, the total file read time includes the following processes:

1. The master distributes filenames to workers;
2. Workers read files and parse files into a map object;
3. Workers send parsed files as long strings back to the master.



Figure 3 Total file read time trend

From Figure 3 it is easy to see that when worker count increases from 1 to 3, the total file read time has a great drop. After that, it goes down a little bit as worker count increases, but the portion of change gets smaller and smaller. This figure can be explained with Ahmdahl's Law. As the file reading and parsing procedures can processed in parallel, more workers working in parallel can decrease the total file reading time. And code that cannot be paralleled, like the process of sending and receiving messages, reading file line by line, and creating the inverted file (it is the case in my design) will limit the performance earned from parallelism.

* 1. Total time building in-memory inverted file

Figure 4 shows the total time in building in-memory inverted file with worker count going up. Specifically, this timing statistics is handled by the master node. Once the master receives a long string representing a serialized inverted book, it resumes the clock as well as begins to merge the serialized inverted book into a map object. After the book merging process is done for current book, it pauses the clock. Finally, the in-memory map object is a representation of all the parsed books.



Figure 4 Total time building the in-memory inverted file as worker count increases

When the worker count goes up from 1 to 7, the total time building the in-memory file keeps steady. This result makes sense since the master node is doing all this work, no matter how many workers it has.

When the worker count goes up to 9 and 11, the total time building the in-memory file begins to shoot up. I am not 100% sure about the reason behind this, but one plausible explanation could be like this.

books[ terms[0] ].push\_back(fileSeq \* VERSES\_PER\_CHAPTER \* CHAPTERS\_PER\_BOOK + std::stoi(terms[i]));

The parsed words and their positions (file #, chapter #, verse #) are recorded in a map object called books. Let’s think about 2 extreme cases: the case of only 1 worker and the case of 11 workers.

When there is only 1 worker, the largest book is first parsed and merged into books, then the second largest book, then the third, and so on. The chances of slot overflow of the map and reallocating memory are small since the merged books will be smaller and smaller.

When there are 11 workers, let’s assume all the workers start to worker at the same time. Then the 11th largest book will be first merged into books, then the 10th largest book, or the 12th largest book, it depends. After many small books are merged, the longest string parsed from the largest file is ready to merge. We expect many more chances of full slots and reallocation for the map books. The memory operations are expensive, which is reflected in total time in building in-memory file increasing in accelerated speed.

* 1. Total time writing the output file

Figure 5 shows the total time writing the output file as worker count goes up. Specifically, this process includes:

1. Sort the vector of integers representing positions (file #, chapter #, verse #) of the word in the original books;
2. Serialize the map object and write out to file with required format



Figure 5 Total time writing the output file as worker count increases

If the worker count is different, the generated vector of integers representing positions of the word could be in different order, which results in different time in sorting the vector of integers, and influences the total time writing the output file. The fluctuation in time is reflected in Figure 5, but the change is not much (note: the scale of y-axis is small).

* 1. Total time for the entire program



Figure 6 Total time for the entire program as worker count increases

Figure 6 shows the total execution time of the program as worker count goes up. It drops quickly from the case of 1 worker to 3 workers, and then the total execution time keeps steady (goes down a little bit) when worker count increases from 3 to 7. After that, the total execution time goes somewhat up. And this result can be explained with Ahmdahl's Law. More workers means more parallelization in file reading and parsing, but eventually, the code that cannot be parallelized (in my design) like message passing, book merging, and file output will limit the performance achieved from parallelization. Also, more workers means more time in building the in-memory inverted file (not strictly right, but roughly). From the timing statistics of the experiments, we can see the shortest execution time is achieved when worker count is 5 or 7.

* 1. Time comparison among different modules



Figure 7 Time comparison among different modules with worker count going up

From Figure 7 we can clearly see that when there is only 1 worker, time spent in file reading and parsing takes most of the execution time. When there are more than 1 workers, the time spent in building in-memory inverted file takes the largest portion, then the time in writing the output file. Neither parts of the code can be parallelized in my design. From this comparison, we see a great speedup results from parallelization, and code that cannot be parallelized limits the total performance achieved by parallelization.

1. Lessons learnt from this project

Workers get project from the master, and after work is done, the master collects the work and do the rest of the work, which, maybe in serial. Actually, this fashion is very like the previous design in hw2 with multi-processes and hw3 with multi-thread.

With MPI, the communication for the master and worker nodes is about pushing messages into buffer, and fetching messages from buffer, and the transport layer of the network will take care of the real communication process. Since most of the very detailed stuff is done under the hood, what the programmer needs to do is just configuring the parameters when calling the APIs.