<OS Project2>

Kim Kyung Min

2016311600

1. Investigation on MapReduce

: First, let's look inside what is the basic concept and goal of MapReduce technique. It is a software framework that Google has made for processing the very large data in the distributed parallel computing environment. The heart of this concept is, this programming allows for massive scalability across large amount of clustered computing. The core function is 'Map' and 'Reduce'. What 'Map' function does is applying the function to all factors in the data. That is, it split the given data and merge into a key/value pair. Next, the 'Reduce' function analyzes the recursive data structure and made some return values by using the passed several commands from 'Map'.

This concept is used in the most popular big data program 'Hadoop' or 'Apache'. Below figure well describe the basic mechanism of the MapReduce programming.

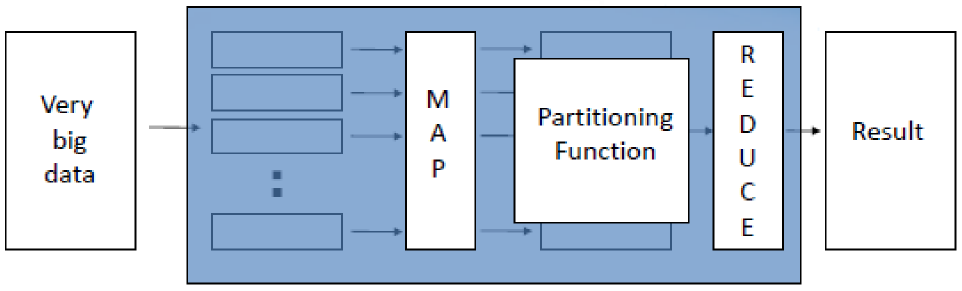


Figure 1. MapReduce on Hadoop (http://elecs.tistory.com/164)

In our project, same concept applies. In the map function, it takes an key/value sets and put them in to the shared data structure. It is just an intermediate form, so the reduce function runs after the map function to do something with the shared data structure. It merges the given values and form a small set of values. I think this mechanism is similar to the divide and conquer algorithm. Both of them breaks one large data to several ones and merge them after processing each piece.

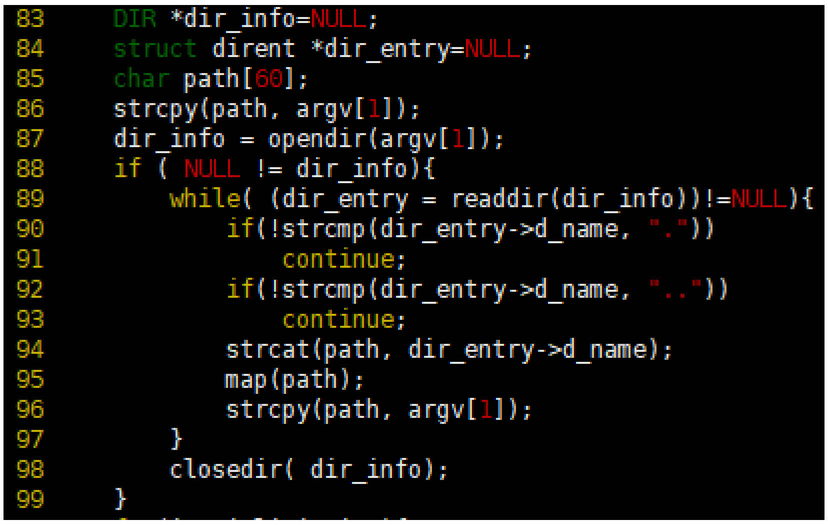
2. Implementation

: This topic will be divided into three parts. Each explains the implementation of 'single_mapreduce.c' or 'multi_mapreduce.c' source file and the common part of two source files. First, let's look inside the common concepts.

1) Common concepts

- Opening the files

  : The one difference between our project and project on Github is that our project passes the name of folder to main function in wordcount, not the name of files. To process this properly, I used strcat and strcpy from string.h header file and used opendir and readdir from dirent.h header file.

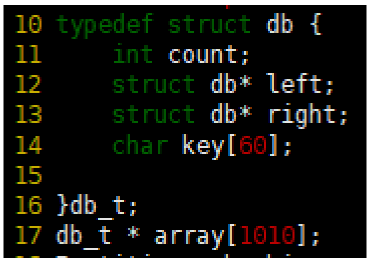


```
83      DIR *dir_info=NULL;
84      struct dirent *dir_entry=NULL;
85      char path[60];
86      strcpy(path, argv[1]);
87      dir_info = opendir(argv[1]);
88      if ( NULL != dir_info){
89          while( (dir_entry = readdir(dir_info))!=NULL){
90              if(!strcmp(dir_entry->d_name, "."))
91                  continue;
92              if(!strcmp(dir_entry->d_name, ".."))
93                  continue;
94              strcat(path, dir_entry->d_name);
95              map(path);
96              strcpy(path, argv[1]);
97          }
98          closedir( dir_info);
99      }
```

  This code is the part of 'single_mapreduce.c' file. We can see that the we are storing the directory information in to dir_info variable using the opendir and the first argument of the main function (line 87). Next, if we can find the folder (line 88), traverse that folder until we search all files (line 89). Note that I exclude the '.' and '..' because they indicate the directory, not the file. I then concatenate the name of file to the folder name then pass it to the mapper function. I iterate this loop to pass all the path of existing files.

- Structure named 'db_t'

: I used structure named 'db' to store the information of each key and value pair.



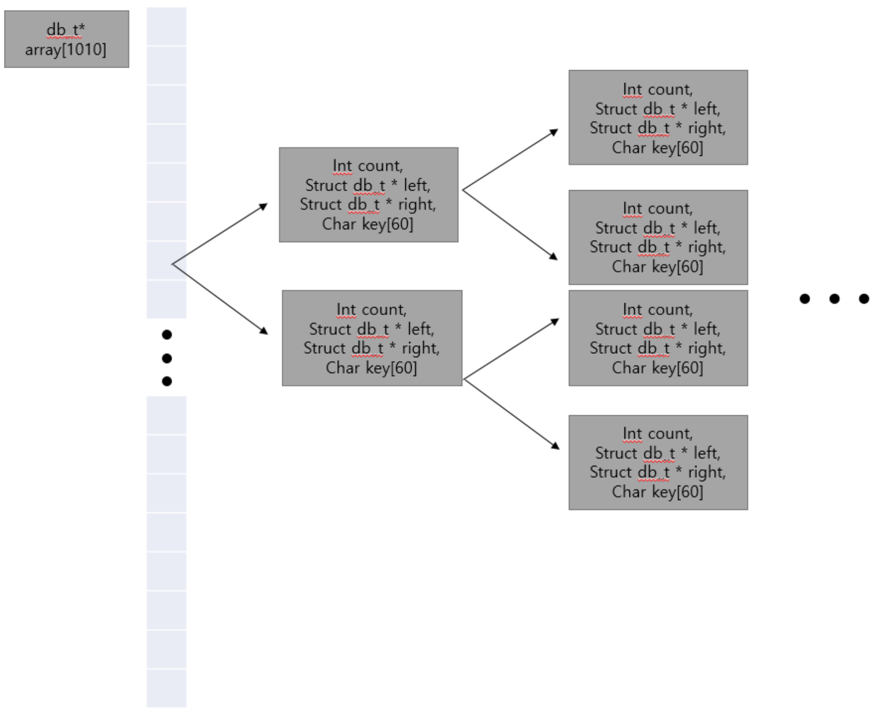
```c
typedef struct db {
    int count;
    struct db* left;
    struct db* right;
    char key[60];

}db_t;
db_t * array[1010];
```

The integer variable count contains the number of key. The defect of my structure is that it did not store the value of pair. It is to maximize the performance, but it violates the basic concept of MapReduce by the way. Two structure pointers are needed because I used the binary search tree in implementing (explained in next part). The character array key stores the key to compare with each input. Finally, I made a structure pointer array to fit the size of partition number. Each index will contain the pointer of root of BST.

- Shared data structure

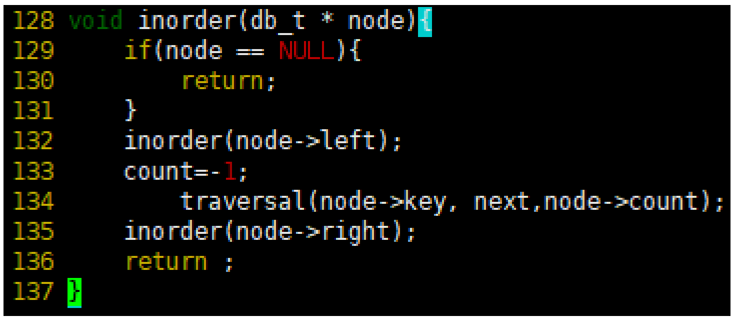
: As I introduced above, I used array of structure pointer and BST to store the key/value pair. The abstract figure is as below.


db_t* array[1010]
Int count, Struct db_t * left, Struct db_t * right, Char key[60]

The MR_Run function in my code processes the adding nodes or key/value pair in the BST. Basically, the result of hashing function acts as an index in my hash table. If the root is NULL, I allocate a new space and insert it as a root. If not, traverse the BST by using strcmp() and insert key/value.

-Sorting

: The reason I used BST is not only its performance is good at inserting or searching(logn), but also supports sorting if I traverse the tree with inorder traversal.



```
void inorder(db_t * node){
    if(node == NULL){
        return;
    }
    inorder(node->left);
    count=-1;
        traversal(node->key, next,node->count);
    inorder(node->right);
    return ;
}
```

You can see that the basic structure is same with common inorder traversal, but difference is that I called Reducer function in the middle of recursive calling.

2) single_mapreduce.c

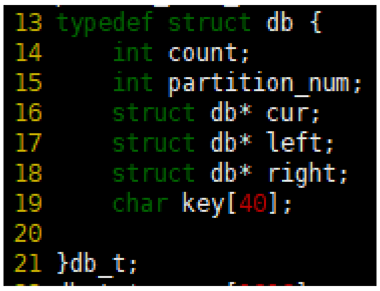
- How I implement Getter function

  : You can see that I used global variable 'count'. It is a kind of trick to reduce the elapsed time. I think the original intend of template code is to access the shared data structure until it can't find more same keys. Rather than, I stored the number of keys in the structure 'db' and store it to the global variable 'count'. I return the character pointer of key to the Reducer function until I consume all counts in the structure.

3) multi_mapreduce.c

- Different structure.

  : In multi-threaded version, the members of 'db' structure are different with the single thread version.



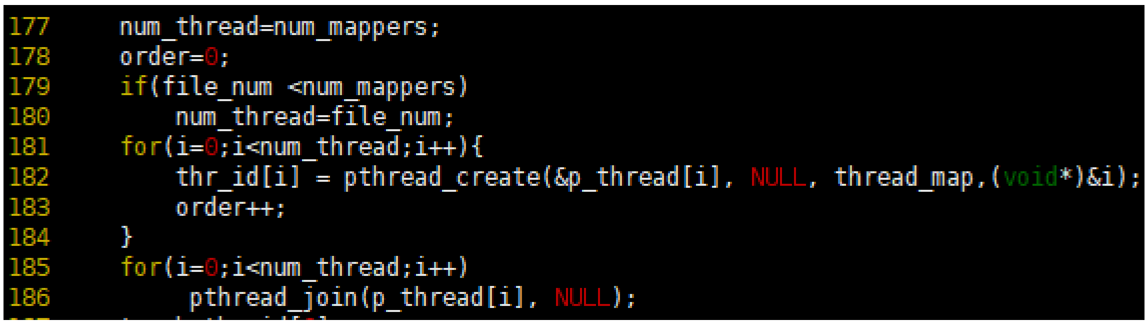
```
typedef struct db {
	int count;
	int partition_num;
	struct db* cur;
	struct db* left;
	struct db* right;
	char key[40];

}db_t;
```

I added integer variable partition_num and structure pointer cur. Two members will be used in implementing the Getter iterator. Specific explanation will be on last part.

- Thread Management

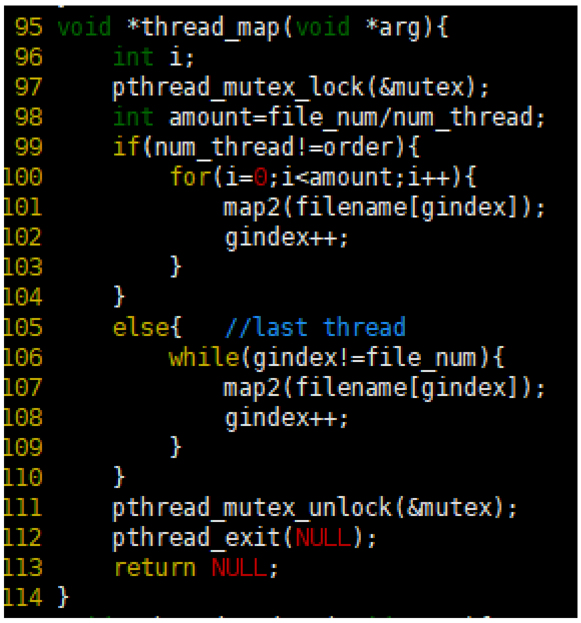
: The hardest constraint was that the number of thread should not exceed the given input as num_mappers or num_reducers. I first used condition variable to manage the number of threads. Right after creating the thread it sleeps and the created thread will send a signal if the number of thread is less than num_mappers.



```
	num_thread=num_mappers;
	order=0;
	if(file_num <num_mappers)
		num_thread=file_num;
	for(i=0;i<num_thread;i++){
		thr_id[i] = pthread_create(&p_thread[i], NULL, thread_map,(void*)&i);
		order++;
	}
	for(i=0;i<num_thread;i++)
		 pthread_join(p_thread[i], NULL);
```

However, the elapsed time increased, so I just made ten threads at the beginning and map the unused file to each thread.

We can see that if the number of files is smaller than num_mappers, we don't have to make more threads. Fitting to the number of files is sufficient. If not, we use num_mappers threads. After creating the threads, I used pthread_join to terminates the thread. Then, let's look inside the thread_map function.

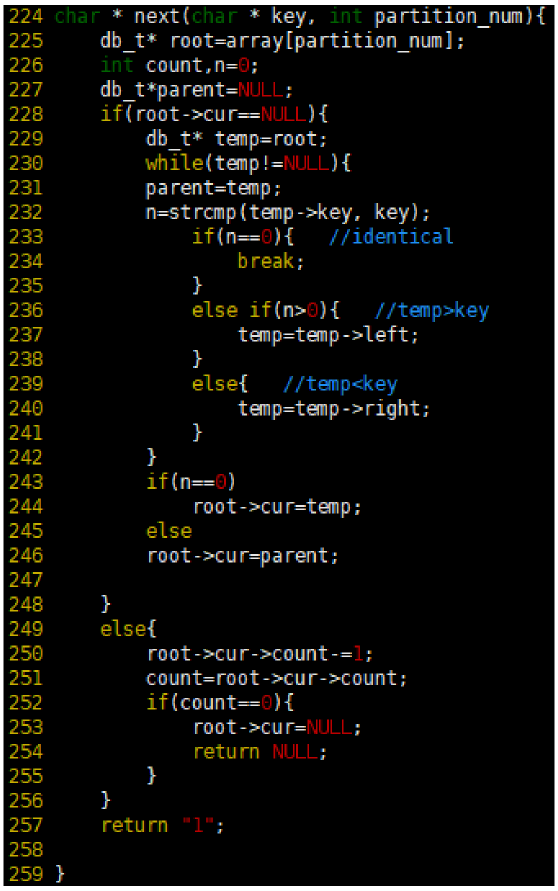


```
void *thread_map(void *arg){
    int i;
    pthread_mutex_lock(&mutex);
    int amount=file_num/num_thread;
    if(num_thread!=order){
        for(i=0;i<amount;i++){
            map2(filename[gindex]);
            gindex++;
        }
    }
    else{   //last thread
        while(gindex!=file_num){
            map2(filename[gindex]);
            gindex++;
        }
    }
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
    return NULL;
}
```

The amount variable is used to assign proper works to each thread. You can see the if-else statement. This statement is used to check whether this thread is the last thread or not. Because I can't assign exactly same amount of jobs to each thread, the last should process all remaining jobs. These whole mechanism is wrapped with mutex locking and unlocking. Because I used several global variables and data structures, locking is important to prevent the interrupts of other threads.

- How I implement Getter function

: I omitted the specific explanation of thread_reduce function because it is almost same with the thread_map function explained in the above part. Actually, the implementation of my Getter iterator is somewhat complicated. Below is the next function in my code.



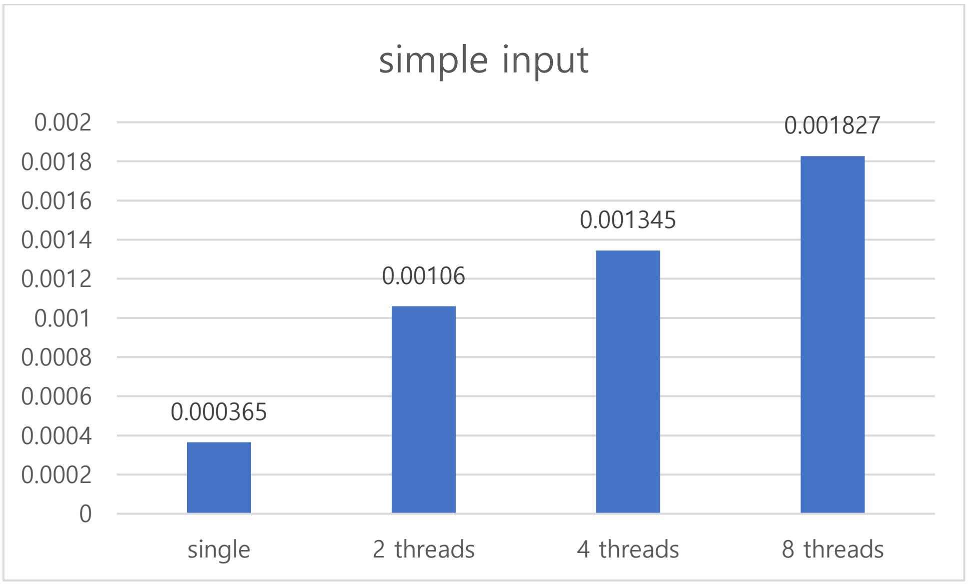
```
224 char * next(char * key, int partition_num){
225     db_t* root=array[partition_num];
226     int count,n=0;
227     db_t*parent=NULL;
228     if(root->cur==NULL){
229         db_t* temp=root;
230         while(temp!=NULL){
231         parent=temp;
232         n=strcmp(temp->key, key);
233             if(n==0){   //identical
234                 break;
235             }
236             else if(n>0){   //temp>key
237                 temp=temp->left;
238             }
239             else{   //temp<key
240                 temp=temp->right;
241             }
242         }
243         if(n==0)
244             root->cur=temp;
245         else
246         root->cur=parent;
247 
248     }
249     else{
250         root->cur->count-=1;
251         count=root->cur->count;
252         if(count==0){
253             root->cur=NULL;
254             return NULL;
255         }
256     }
257     return "1";
258 
259 }
```

Here, the partition_num and cur pointer members in my new db structure are used. Because the template code returns nothing from the Reducer function, we can't know the current node's address. To store this information, I stored where my current position is to the root node's cur variable. Line 228 indicates that process. If the root does not contain the current position of node, it finds from the BST and store it to root's cur pointer. By doing so, next accessing can use the cur pointer of the root (else case) and can directly access to currently processed node, not need to search in the BST again. This will improve the performance dramatically because probably more than 1 million words are stored in the long input folder. Remaining codes are same with the single-threaded case, so I'll omit the explanation.

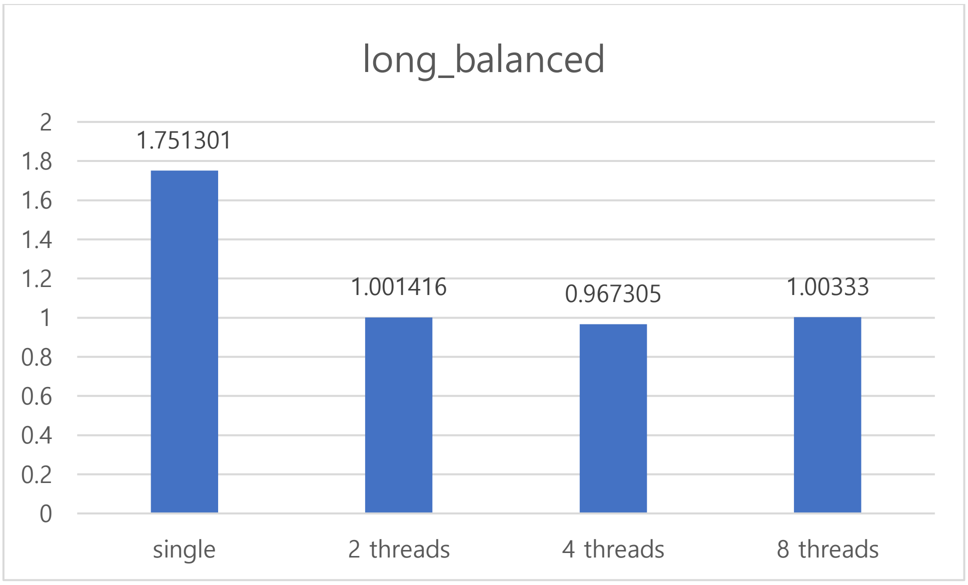
3. Project result

: I divided the result case in to two cases. One for the single-threaded version and the other for multi-threaded version.

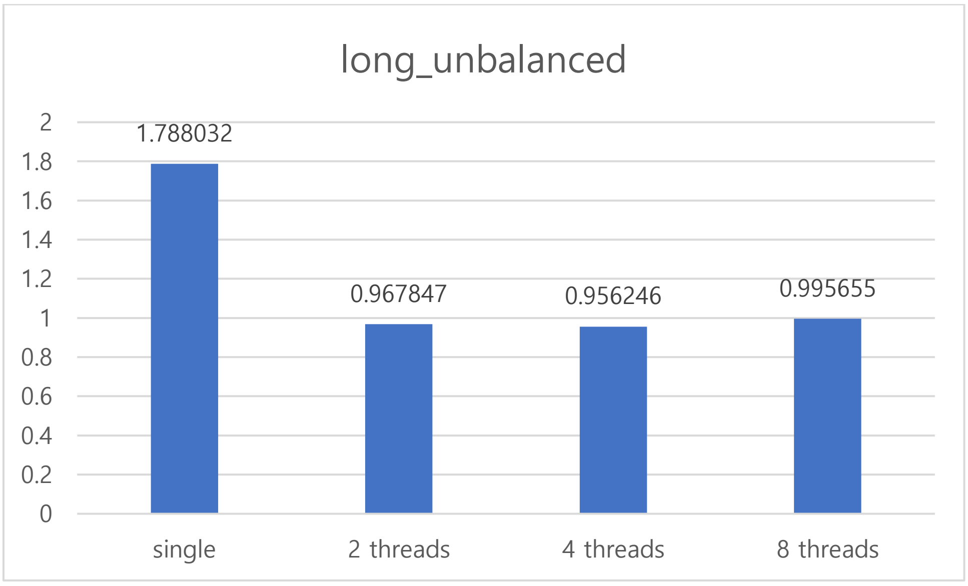
1) Simple Input



2) Long_balanced Input



3) Long_unblanced Input



4) Analysis

➔ We can see that in the simple input case, the variation of elapsed time is as below. As the graph shows, it shows the linear increase.

Simple input	Single thread	2 threads	4 threads	8 threads
Elapsed time	0.000365	0.00106	0.001345	0.001827

The time increased even in the multi-threaded versions due to the thread creating overhead. Bigger the thread number, longer the elapsed time takes (bigger overhead). This is because the input size is small, thus the thread creating overhead is bigger than the processing of each input.

However, in the long input cases when the input size is much larger than the simple input case, we can see the significant performance improvement. In the single-threaded version, the elapsed time is around 1.7 seconds, but it reduced to around 1.0 seconds in the multi-threaded cases. I don't know the exact reason why the performance does not improve more as the number of thread increases.

5) Reason why single version is faster than the multi-threaded version sometimes

: I ran this program for 20 or more times and sometimes, the single threaded version is faster than the multi-threaded version. I think this is because the specific ways of implementing are different. Other parts are almost same between single and multi version, but the implementing of Reducer function is different. In single-threaded version, I only used a global variable ‘count’ to store the duplicated keys in order to maximize the performance. However, in the multi-threaded version, I used extra variables because I have to handle the concurrency issues and thus have to store the current position of node in the root node, not in the global variable. Those additional works result in the extra elapsed time in some trials.