CMPE 300 Application Project Documentation

Course ID: CMPE300
Course Name: Algorithm Analysis
Name of the Student: Baran Kılıç
Project title: MapReduce
Type of the project: Programming project
Submission date: 18.12.2017

Introduction

In this project, we need to count the occurrences of words in a file. If this file is very large, a sequential algorithm to solve this problem will consume to much time. Therefore, to solve this problem, we use a parallel algorithm named MapReduce. This algorithm works in several steps. First, it splits the input. Then the input is mapped and the sorted. Lastly, it is reduced and we obtain the result. As communication protocol for parallel computing, we use Message Passing Interface (MPI).

Program Interface

I wrote my code on Ubuntu and used MPICH and C++. To install MPICH, you need to execute the command "sudo apt-get install mpich" on terminal on Ubuntu. To compile the program, you need to execute the command "mpicxx main.cpp". The program can be run using "mpirun -n 4 ./a.out". The number after to option "n" specifies the number of processes to use.

If you run the program, it reads "speech_tokenized.txt" and outputs the result to the file "output.txt" and then ends.

Program Execution

The program is supplied with a input file that contains some words in it. The program outputs a file that contains the words and the number of occurrences of this word in the input file. The programs count the number of occurrences of each word.

Input and Output

The input file we provide have a special format. The words must be tokenized and every word must be on separate lines. The input file should have the name "speech_tokenized.txt" (without quotes).

Input example:
my
fellow
citizens
events
12
my
my
citizens

The output file has the name "output.txt". It contains a distinct word on each line and the number of occurrences of this word besides it separated by a space. The words are sorted in increasing order.

Output example:
12 1
citizens 2
events 1
fellow 1
my 3

Program Structure

I used MPI in this project to communicate between processors. There is one master process and are several slave processes. I used blocked send/receive for communication.

First, the input words are partitioned into segments. Then each segment is sent to a distinct process and each process counts its word occurrences which corresponds to the mapping stage. The information is stored in a structure. Later, this structure list is put together and sorted. Last step is reducing, which outputs the total count of each word.

To send the words together with its frequency in text, I created a structure. It contains a integer to store the frequency and a char array to store the word. I selected a maximum word length to ease the implementation (fixed sized word length means that I do not have to send also the word length).

In MPI, you can send predefined data types. These are the variables types in C. For example MPI_INT for integers, MPI_CHAR for characters. To send a structure, you need to define your own MPI data type. I created a new data type for my structure.

In the master process, I read the input file and stored the words in the file in a 2 dimensional char array. The first index of the array points to a word. The second index of the array points to the individual characters in a word.

I partitioned this word array by hand. By dividing the total word count by slave processes, I found the average word count per process. There is a remainder part of this division. I assigned the remainder part to the last processor.

I send the number of data to be send to each slave process. I send the corresponding data to each slave process using blocking send. Each slave process receives its word list. For each word, it creates a "word_count" structure and initializes its word field with word and count field with 1. In the end, the slave has an array of structures. Each slave send this mapped data to the master process. The master process receives the data from each slave and stores the data in an array of structures. The master process distributes this data to the slaves again like I explained before. The slaves receive the data and sort the data with merge sort. The slave processes send the the sorted data to the master. The master also uses merge sort for the last merge. Since the data is sorted, the master process looks to the consecutive repetitions in the data and count the repetitions. The count together with the word is stored in a separate array of structures. This is the reducing part. Lastly, the master process write this words and their counts to a file.

Examples

Input file:
my
my
car

Splitting the words: Let the number of slave processes be 2. 3/2=1 and 1 remainder. 1 word for each process and additionally the remainder is for the last process.
Slave 1:
my

Slave 2:
my
car

After mapping:
Slave 1:
my 1

Slave 2:
my 1
car 1

Master process collects the words:
my 1
my 1
car 1

Master splits data:
Slave 1:
my 1

Slave 2:
my 1
car 1

Slave processes sorts the data:
Slave 1:
my 1

Slave 2:
car 1
my 1

Master process collects the words:
Master:
my 1
car 1
my 1

Master does the last sorting:
car 1
my 1
my 1

Master reduces:
car 1
my 2

Final Result:
car 1
my 2

Improvements and Extensions

Since the structure of this project is straightforward, I cannot think of any improvements. Maybe, we can try to improve the performance. But, there is only send and receive functions. Therefore, there cannot be much improvement in my opinion.

Difficulties Encountered

Understanding the way of sending data in MPI_Send was a big problem for me. It only supports C data types and I need to send an array of words. First, I understand that I can send a char using MPI_CHAR data type. Then, I saw that I can send multiple data at once. This means that I can send a char array which corresponds to a word. But, I need to send several words to an array. Sending each word with an separate send function would decrease the performance because the communication is the bottleneck in parallel algorithms. I had to find another way. I thought that I can concatenate the words and send it as one string and split the words in the slave process. But this solution also complicated the situation and I had to do unnecessary string operations. In the end, I found the correct way. The array that I use to store the words is a 2 dimensional char array and the words are stored consecutively. The MPI_Send requires the beginning of the data and the lenght of the data. Therefore, if I give the beginning of a word as the start address for the function and specify the length of the data as max word length multiplied with how many words I want to send, it will send the data in the way that I want to send.

My second problem was sending a structure using MPI_Send. Since it only supports C data types, I had to define my own MPI data type to send a struct. First, I couldn't understand how can I do this. Then, I found out that I can define a data type but there wasn't any examples that explained how to define this new data type. After some searches, I found a good example that explained how to specify the arguments to create a new data type. It showed how to set block numbers, lengths and displacements.

Conclusion

I used MPI with C for this project and implemented MapReduce algorithm. I used C++ as programming language to facilitate reading input from a file and writing output to file but I used C bindings for MPI. Since I also used C, I refreshed my knowledge about C. I learned many concepts in MPI.

Appendices

Source Code:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <mpi.h>

using namespace std;

static const int max_word_length = 50;

// structure to store the frequency of the words
struct word_count {
    char word[max_word_length]; // string to store the word
    int count; // the word's frequency
```

```
};

void Merge(struct word_count arr[], int low, int mid, int high) {
    int mergedSize = high - low + 1;
    struct word_count temp[mergedSize];
    int mergePos = 0;
    int leftPos = low;
    int rightPos = mid + 1;

    while (leftPos <= mid && rightPos <= high) {

        if (strcmp(arr[leftPos].word, arr[rightPos].word) < 0) {
            temp[mergePos++] = arr[leftPos++];
        }
        else {
            temp[mergePos++] = arr[rightPos++];
        }
    }

    while (leftPos <= mid) {
        temp[mergePos++] = arr[leftPos++];
    }

    while (rightPos <= high) {
        temp[mergePos++] = arr[rightPos++];
    }

    for (mergePos = 0; mergePos < mergedSize; ++mergePos) {
        arr[low + mergePos] = temp[mergePos];
    }
}

// Classical MergeSort
void MergeSort(struct word_count arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        MergeSort(arr, low, mid);
        MergeSort(arr, mid + 1, high);
        Merge(arr, low, mid, high);
    }
}

int main(int argc, char *argv[]) {
    /********** Setting up MPI **********/
    MPI_Init(&argc, &argv); // Initialize the MPI environment

    MPI_Status status; // stores the source of the message and the tag of the
message for receive operation

    int data_tag = 1; // tag used as a argument while sending message. its value
is unimportant for this project

    int my_id; // my process id
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id); // sets the value of my_id
    int num_of_procs; // how many processes were started
    MPI_Comm_size(MPI_COMM_WORLD, &num_of_procs); // sets the value of
num_of_procs

    int master_process = 0; // id of the master process

    /*
     * int MPI_Type_create_struct(
     *   int count,
     *   int array_of_blocklengths[],
```

```
     *   MPI_Aint array_of_displacements[],
     *   MPI_Datatype array_of_types[],
     *   MPI_Datatype *newtype
     * );
     *
     * count: [in] number of blocks (integer)
     * array_of_blocklength: [in] number of elements in each block (array of
integer)
     * array_of_displacements: [in] byte displacement of each block (array of
integer)
     * array_of_types: [in] type of elements in each block (array of handles to
datatype objects)
     * newtype: [out] new datatype (handle)
     */

    /********** Creating data type for struct **********/

    MPI_Datatype word_count_datatype; // new MPI data type to send "word_count"
structs
    int wc_block_num = 2; // number of variables in "word_count" struct
    int wc_block_len[2] = {50,1}; // number of elements of the variables in
"word_count" struct. 50 for char , 1 for int
    MPI_Aint wc_displacement[2]; // byte displacements of each variable in
struct
    wc_displacement[0] = offsetof(struct word_count,word);
    wc_displacement[1] = offsetof(struct word_count,count);
    MPI_Datatype wc_types[2] = { MPI_CHAR, MPI_INT}; // data types of the
variables in "word count" struct

    // creates the new data type

MPI_Type_create_struct(wc_block_num,wc_block_len,wc_displacement,wc_types,&word_
count_datatype);
    MPI_Type_commit(&word_count_datatype); // commits new data type to the
system


    /********** Master process **********/
    if(my_id == master_process){ // this process is the master process

        /********** Reading the input **********/
        ifstream input_file("speech_tokenized.txt"); // input file stream to
read the input file

        // counts the number of newlines in the input file (iterates over input
stream buffer and counts the number of '\n')
        int line_num =
count(istreambuf_iterator<char>(input_file),istreambuf_iterator<char>(),'\n');

        // since we iterated over input, we have to reset the pointer that show
the next input
        input_file.clear(); // clear fail and eof bits of file stream
        input_file.seekg(0, ios::beg); // sets position of the next character of
the stream to the beginning of the file

        string line; // string to store input lines
        char word_array[line_num][max_word_length]; // array of strings to store
words
        for(int i=0;i<line_num;i++){ // for each line in input get the line and
store the word in the array
            input_file >> line;
            strcpy(word_array[i],line.c_str()); // copies word to the array
        }
        input_file.close();
```

```
        /********** Distributing words to slaves **********/
        int number_of_slaves = num_of_procs - 1; // number of slave processes
        int average_data_per_process = line_num / number_of_slaves;

        for(int i=0;i<number_of_slaves;i++){
            int data_start = i*average_data_per_process; // start index of the
data to be send
            int data_end = (i+1)*average_data_per_process - 1; // end index of
the data to be send

            // when the data is not divided evenly by number of slaves, we
assign the remaining data to the last process
            if(i == number_of_slaves-1){
                data_end = line_num-1;
            }
            int num_of_data_to_send = data_end - data_start + 1;
            int target_id = i+1; // id of the process that will send the message
to

            // sends the size of the data to be send

MPI_Send(&num_of_data_to_send,1,MPI_INT,target_id,data_tag,MPI_COMM_WORLD);
            // sends the words to the slave

MPI_Send(&word_array[data_start],num_of_data_to_send*max_word_length,MPI_CHAR,ta
rget_id,data_tag,MPI_COMM_WORLD);
        }

        /********** Collecting mapped words from slaves **********/
        struct word_count word_count_array[line_num]; // struct array to store
the mapped words
        for(int i=0;i<number_of_slaves;i++){
            int data_start = i*average_data_per_process; // start index of the
data to be received
            int data_end = (i+1)*average_data_per_process - 1; // end index of
the data to be send
            // when the data is not divided evenly by number of slaves, we
assign the remaining data to the last process
            if(i == number_of_slaves-1){
                data_end = line_num-1;
            }
            int num_of_data_to_receive = data_end - data_start + 1;
            int slave_id = i+1; // id of the process from which we will receive
the message

            // we get mapped words and store them in word_count_array

MPI_Recv(&word_count_array[data_start],num_of_data_to_receive,word_count_datatyp
e,slave_id,data_tag,MPI_COMM_WORLD,&status);
        }

        /********** Distributing mapped words to slaves to be sorted **********/
        for(int i=0;i<number_of_slaves;i++){
            int data_start = i*average_data_per_process; // start index of the
data to be send
            int data_end = (i+1)*average_data_per_process - 1; // end index of
the data to be send
            // when the data is not divided evenly by number of slaves, we
assign the remaining data to the last process
            if(i == number_of_slaves-1){
                data_end = line_num-1;
            }
            int num_of_data_to_send = data_end - data_start + 1;
```

```
            int target_id = i+1; // id of the process that will send the message
to

            // sends the size of the data to be send

MPI_Send(&num_of_data_to_send,1,MPI_INT,target_id,data_tag,MPI_COMM_WORLD);
            // sends the mapped words to the slave

MPI_Send(&word_count_array[data_start],num_of_data_to_send,word_count_datatype,t
arget_id,data_tag,MPI_COMM_WORLD);
        }

        /********** Collecting sorted mapped words from slaves **********/
        struct word_count word_count_array2[line_num]; // struct array to store
the sorted mapped words
        for(int i=0;i<number_of_slaves;i++){
            int data_start = i*average_data_per_process; // start index of the
data to be received
            int data_end = (i+1)*average_data_per_process - 1; // end index of
the data to be send
            // when the data is not divided evenly by number of slaves, we
assign the remaining data to the last process
            if(i == number_of_slaves-1){
                data_end = line_num-1;
            }
            int num_of_data_to_receive = data_end - data_start + 1;
            int slave_id = i+1; // id of the process from which we will receive
the message

            // we get sorted mapped words and store them in word_count_array2

MPI_Recv(&word_count_array2[data_start],num_of_data_to_receive,word_count_dataty
pe,slave_id,data_tag,MPI_COMM_WORLD,&status);
        }

        MergeSort(word_count_array2,0,line_num-1); // the last merge for
partially sorted "word_count" structs

        /********** Reducing the list **********/
        vector<word_count> reduced_list; // "word_count" vector to store the
reduced list
        if(line_num > 0){ // if there is data
            int count = 1; // count of the word. initially equal to the count of
the first word
            char word[50]; // char array to store a word. initially equal to the
first word
            strcpy(word, word_count_array2[0].word);

            for(int i=1;i<line_num;i++){
                if(strcmp(word,word_count_array2[i].word) == 0){ // if the next
word is the same increment the count
                    count++;
                }else{ // if the next word is different add the word to the
reduced list
                    word_count wc;
                    strcpy(wc.word,word);
                    wc.count = count;
                    reduced_list.push_back(wc);
                    count = 1; // update the count for this new word
                    strcpy(word,word_count_array2[i].word); // update the string
for this new word
                }
            }
            word_count wc;
```

```
            strcpy(wc.word,word);
            wc.count = count;
            // add the last word we counted to the reduced list since we do not
check it in the for loop above
            reduced_list.push_back(wc);
        }

        /********** Writing the word counts to a file **********/
        ofstream outFile("output.txt"); // output stream to write the data
        for (auto &i : reduced_list) { // iterates over the reduced list and
outputs to the file
            outFile << i.word << " " << i.count << endl;
        }

    }else{
        /********** Slave process **********/

        /********** Receiving the splitted list of words from master process
**********/
        int num_of_data_to_receive;
        // we get the number of data to be received from the master process

MPI_Recv(&num_of_data_to_receive,1,MPI_INT,master_process,data_tag,MPI_COMM_WORL
D,&status);

        // string array to store splitted word array
        char slave_word_array[num_of_data_to_receive][max_word_length];

MPI_Recv(&slave_word_array[0],num_of_data_to_receive*max_word_length,MPI_CHAR,ma
ster_process,data_tag,MPI_COMM_WORLD,&status);

        /********** Mapping the words **********/
        struct word_count word_count_array[num_of_data_to_receive]; // struct
array to store the mapped words
        for(int i=0;i<num_of_data_to_receive;i++){
            strcpy(word_count_array[i].word, slave_word_array[i]);
            word_count_array[i].count = 1;
        }

        /********** Sending the mapped words to master process **********/

MPI_Send(&word_count_array[0],num_of_data_to_receive,word_count_datatype,master_
process,data_tag,MPI_COMM_WORLD);

        /********** Receiving the mapped words that are to be sorted from master
process **********/

MPI_Recv(&num_of_data_to_receive,1,MPI_INT,master_process,data_tag,MPI_COMM_WORL
D,&status);
        struct word_count word_count_array2[num_of_data_to_receive]; // struct
array to store the mapped words that are to be sorted

MPI_Recv(&word_count_array2[0],num_of_data_to_receive,word_count_datatype,master
_process,data_tag,MPI_COMM_WORLD,&status);

        // sorts the array
        MergeSort(word_count_array2,0,num_of_data_to_receive-1);

        /********** Sending the sorted mapped words to master process
**********/

MPI_Send(&word_count_array2[0],num_of_data_to_receive,word_count_datatype,master
_process,data_tag,MPI_COMM_WORLD);
```

```
    }
    MPI_Type_free(&word_count_datatype);  // free datatype when done using it
    MPI_Finalize();  // terminates the MPI execution environment
    return 0;
}
```