# Introducing Students to MapReduce with Phoenix++ Documentation

**CSinParallel Project**

Aug 15, 2016

CONTENTS

**Last Updated:** 2016-08-15

Welcome to the module for introducing students to MapReduce with Phoenix++! The first section, *What is MapReduce?* introduces students to the overall paradigm of MapReduce, discusses the history of Phoenix++ MapReduce, and how it differs from Hadoop and Google MapReduce. The next section, *Counting Pease with MapReduce* discusses how to use MapReduce to count words in a file, using a famous poem from Mother Goose. The third section, entitled *Getting Started with Phoenix++*, explores the Phoenix++ code base, discusses files required for custom implementations, and gives an overview of the word count example provided by the framework. Contents:

CHAPTER

ONE

# WHAT IS MAPREDUCE?

## 1.1 Motivation

In today's world, multicore architectures are ubiquitous. But, the majority of programs that people write are still serial. Why is that? While some people may be unaware that they can leverage the multiple cores on their computers, the truth is that parallel computing is very difficult. In many cases, the programmer must consider many factors that have nothing to do with problem he or she is trying to parallelize. For example, to implement a program in Pthreads, a programmer must physically allocate, create and join any threads they want to use. The programmer must also be aware of data races, and use synchronization constructs as necessary.

This is not unique to Pthreads. In MPI for example, you have to explicitly specify what messages you want to send to what node, and how to synchronize messages. As you can imagine, this creates a lot of overhead for the programmer. As those who have programmed previously in Pthreads, MPI, or OpenMP can attest, debugging parallel programs can be very difficult! When things execute in parallel, they execute *non-deterministically*. This non-determinism can cause a lot of headaches.

As multicore and parallel systems became more prevalent, computer scientists began to ask the question if parallel computing is harder that it needs to be. Some libraries such as OpenMP "hide" some of the work required with threads through the use of pragmas. Hiding implementation details is known as abstraction. However, even with abstraction, the programming still has to worry a lot about the "management" aspects of the program. Furthermore, similar applications can be parallelized in the way. Researchers began exploring ways to create an automated framework for parallel computing.

## 1.2 Enter MapReduce

In 2004, Jeffrey Dean and Sanjay Ghemawhat of Google published a paper on the MapReduce paradigm. Google uses MapReduce as the backbone of its search engine, and uses it for multiple operations. It is important to note that Google did *not* invent MapReduce; the paradigm has existed for decades in functional languages. However, the paper's release was a watershed moment in parallel computing, and spelled the beginning of an upsurge in interest in the paradigm that has led to many innovations over the last decade.

Google's implementation of MapReduce is closed source and proprietary. In 2006, work on the Hadoop project was started by Doug Cutting, an employee of Yahoo!. Hadoop is named after a plush toy elephant belonging to Cutting's son, and the eponymous elephant features prominently in the Hadoop logo. Over the last six years, Hadoop has been widely adopted by many tech giants, including Amazon, Facebook and Microsoft.

It is important to note that both Google's implementation of MapReduce and Hadoop MapReduce were designed for very large datasets, on the order of hundreds of gigabytes and petabytes. The goal is to efficiently streamline the processing of these large numbers of documents by distributing them over thousands of machines. Note that for smaller datasets, the system may have limited benefit; the Hadoop Distributed File System (HDFS) can prove to be a

bottleneck. However, the concept of MapReduce is still very attractive to programmers with smaller datasets or more limited computational resources, due to its relative simplicity.

**Note:** Want to play with a Hadoop system on the web? Check out WebMapReduce! Access the module at this link.

## 1.3 Phoenix and Phoenix++

In 2007, a team at Stanford University led by Christos Kozyrakis began exploring how to implement the MapReduce paradigm on multi-core platform. Their thread-based solution, Phoenix, won best paper at HPCA'07, and has been cited over 900 times. An update on Phoenix (Phoenix 2) was released in 2009. In 2011, Phoenix++ was released. A complete re-write of the earlier Phoenix systems, Phoenix++ enables development in C++, and significantly modularizes and improves the performance of the original code base.

**Note:** This entire module is based on Phoenix++ the latest release of Phoenix. Please note that if you are interested in using the earlier modules, these instructions may not directly apply.

CHAPTER

TWO

# COUNTING PEASE WITH MAPREDUCE

## 2.1 Word Frequency

We will discuss a classic example called Word Frequency, or Word Count. In this example, the goal is to identify the set of unique words in a text file and compute their associated "counts" or frequencies. Consider the follow poem by Mother Goose:

```
Pease-porridge hot
Pease-porridge cold
Pease-porridge in the pot
Nine days old
Some like it hot
Some like it cold
Some like it in the pot
Nine days old
```

If we were to count the word frequencies in this file, we may get output that looks like the following:

```
cold : 2
days : 2
hot : 2
in : 2
it : 3
like : 3
nine : 2
old : 2
pease-porridge : 3
pot : 2
some : 3
the : 2
```

Notice that each word is associated with the frequency of its occurrence in the poem.

## 2.2 Solving the Problem Using MapReduce

In MapReduce, the programmer is responsible for mainly writing two serial functions: `map()` and `reduce()`. The framework takes care of running everything in parallel. The components of the system are as follows:

- The `map()` function takes a chunk of input, processes it, and outputs a series of *(key, value)* pairs. All instances of the `map()` function (mappers) run independently and simultaneously. This is known as the Map phase.

- A Combiner function sorts all the *(key,value)* pairs coming from the Map phase. The combiner uses a hashing function to aggregate all the values associated with a particular key. Thus, the ouput from the combiner is a series of *(key,list(value))* pairs.
- The `reduce()` function takes *(key, list(value))* pairs and performs a *reduction operation* on each. A reduction operation is any operation that combines the values in some way. The output is a final *(key, value)* pair, where the value is the result of the reduction operation performed on the list. Each instance of the reduce() function (reducer) run independently and in parallel.

So how do we calculate Word Frequency with MapReduce? In the following example, we have three mappers and three reducers. For simplicity, we assume that the file is split on new lines (`\n`) although this need not always be the case. Each mapper takes its assigned chunk of text and splits it into words, and emits *(key,value)* pairs where the key is an individual word, and the value is `1`. If multiple instances of a word are assigned to the same mapper, the local frequencies can be added and emitted instead.

Below, we have an illustration of the Map phase of the algorithm. Observe that the first mapper is emitting a single *(key,value)* pair of (`Pease-porridge,3`) instead of three instances of the pair (`Pease-porridge, 1`). Notice that all mappers run in parallell. This assumes that a local combination operation is occuring.

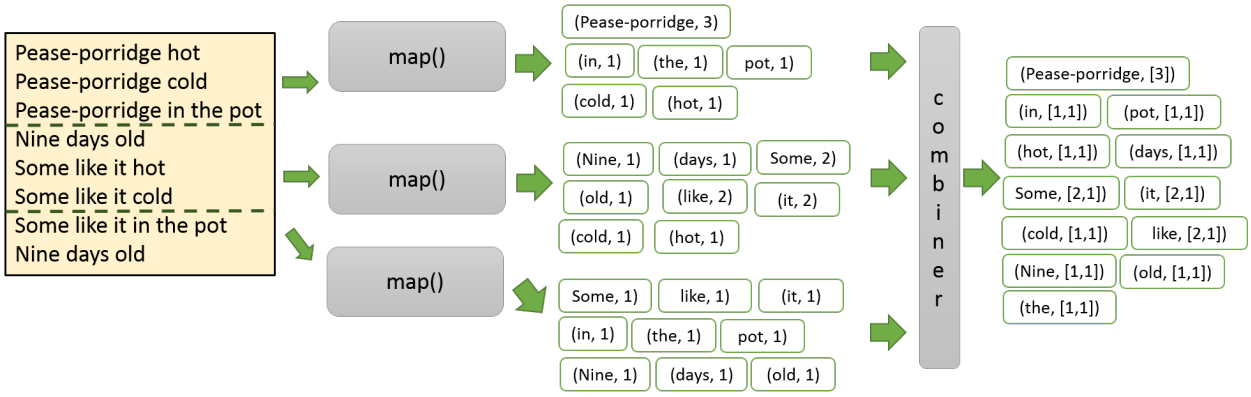


Fig. 2.1: Figure 1: How the Map Phase of the algorithm works.

The combiner acts as a synchronization point; all the mappers must finish prior to the combiner finishing execution. The combiner constructs *(key,list(value))* pairs from the output from the mappers. For example, mapper 2 produced the *(key,value)* pair (`it, 2`), while mapper 3 produced the *(key,value)* pair (`it, 1`). The combiner will aggregate these two pairs and output (`it, [2,1]`).

After the combiner finishes executing, the *(key,list(value))* pairs go to to the reducers for processing. We refer to this as the Reduce phase. The figure below illustrates the Reduce phase for this example. Each reducer gets assigned a set of *(key,list(value))* pairs. For each pair, it performs a reduction operation. In this case, the reduction operation is addition; all the values in the list are simply added together. For example, reducer 2 reduces the pair (`Some, [2,1]`) to (`Some, 3`).

---

**Note:** One thing we do not discuss here is *fault tolerance*. Fault tolerance is most important for large distributed systems. When you have that many computers networked together, it's likely that some subset of them will fail. Fault tolerance allows us to recover from failures on the fly. In the case of Google's Mapreduce, fault tolerance is maintained by constantly pinging nodes. If any node stays silent for too long, the framework marks that node as being "dead", and redistributes its work to other worker nodes. Phoenix and Phoenix++ both have fault tolerance protections. Phoenix++ has an optional execution mode that enables a user to skip data records in the case of segmentation faults and bus errors. This can be invoked through the use of the signal handler.

---

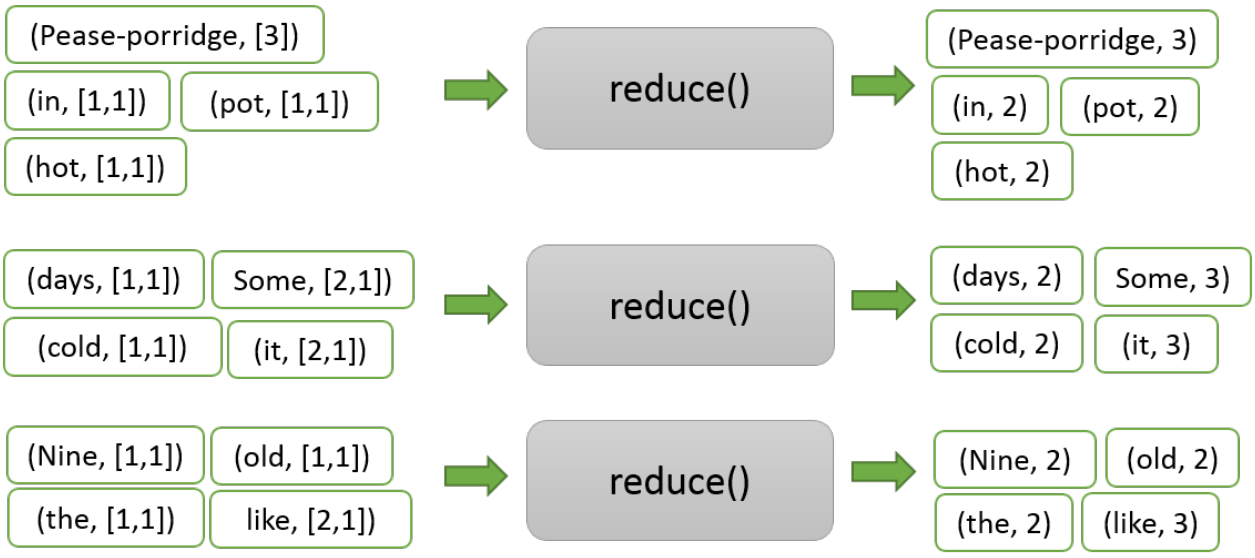


Fig. 2.2: Figure 2: How the Reduce Phase of the algorithm works.

CHAPTER

THREE

# GETTING STARTED WITH PHOENIX++

## 3.1 Introduction

In this section, we will discuss the Phoenix++ wordcount example in detail. You will need a basic knowledge of C/C++ to follow along with these concepts. You can download a copy of the Phoenix++ word count example here. We will start by looking at the file `word_count.cpp`. At the top of the file, there are three structs we should pay attention to:

```
// a passage from the text. The input data to the Map-Reduce
struct wc_string {
    char* data;
    uint64_t len;
};

// a single null-terminated word
struct wc_word {
    char* data;

    // necessary functions to use this as a key
    bool operator<(wc_word const& other) const {
        return strcmp(data, other.data) < 0;
    }
    bool operator==(wc_word const& other) const {
        return strcmp(data, other.data) == 0;
    }
};


// a hash for the word
struct wc_word_hash
{
    // FNV-1a hash for 64 bits
    size_t operator()(wc_word const& key) const
    {
        char* h = key.data;
        uint64_t v = 14695981039346656037ULL;
        while (*h != 0)
            v = (v ^ (size_t)(*(h++))) * 1099511628211ULL;
        return v;
    }
};
```

These three structs define the type of our input chunk (`wc_string`), our keys (`wc_word`) and the hash function that we are going to use to aggregate common keys (`wc_word_hash`). The `wc_string` struct has a data pointer field

(which points to to the start of the chunk), and a `len` field which indicates the size of the chunk. The `wc_word` struct contains only a pointer to the start of a word, along with the defintions on how to compare two "words". At this point, you may be asking yourself, *but, how do you know where a word ends?* Be patient; when we get to the main body of code, it will all become clear. The last struct contains only an operator definition for `()`, which requires a key of type `wc_count` as its single parameter. This is an implementation of the Fowler-Noll-Vo hash function.

While other hash functions can be used, it is best just to leave this code alone.

## 3.2 WordsMR: The word count class

For every application you write using Phoenix++, you will need to define a class for it. Let's start by taking a look at the class header:

```
class WordsMR : public MapReduceSort<WordsMR, wc_string, wc_word, uint64_t,
                                      hash_container<wc_word, uint64_t, sum_combiner, wc_word_hash>
                                      >
```

The first thing to note about this definition is that `WordsMR` is derived from class `MapReduceSort`, which is defined in `mapreduce.h`. This is the primary beauty of Phoenix++; to write your own MapReduce programs, you simply overload the default function defined in the base class. We define the class with the following parameters (in order):

- the implemented class type (*Impl* = `WordsMR`)
- the input data type (*D* = `wc_string`)
- the key type (*K* = `wc_word`)
- the value type (*V* = `uint_64`)
- the definition of the hash container (`hash_container<...>`)

`hash_container` defines the parameters for the hashtable used to aggregate *(key, value)* pairs. A full definition of the `hash_container` class can be found in `container.h`. It's input parameters are:

- the key type (*K* = `wc_word`)
- the value type (*V* = `uint_64`)
- the type of combiner function (*Combiner* = `sum_combiner`)
- The hash function to use (*hash* = `wc_word_hash`)

Note the use of the combiner `sum_combiner`, an associative combiner implemented in Phoenix++. This means, as collisions in our hash table occur, the values are added together. This actually eliminates the need for a reduce function in our application! The other type of combiner is known as the `buffer_combiner`, and reflects typical MapReduce behavior. This combiner chains all the values together. The functions shown below are ones that are commonly overloaded when creating a Phoenix++ MapReduce class:

| Function Name | Description |
|---|---|
| `map()` | Defines the functionality for map tasks. The default definition of the `map()` function is empty. This function must be overloaded for every user-defined MapReduce class. |
| `reduce()` | Defines the functionality for the reduce tasks. By default, the `reduce()` function generates a list of *(key,value)* pairs from a given key and *list(value)* input. |
| `split()` | Defines how input data should be chunked. The default `split()` function returns `0`. This function must be overloaded for every user-defined MapReduce class. |
| `locate()` | Indicates where to access the input data from. By default, the `locate()` function casts the input data as a void pointer. |

For the word count application, the `map()`, `locate()` and `split()` functions are overloaded as public methods.

The class declares a number of global variables, that will be initialized by user input:

```
char* data;
uint64_t data_size;
uint64_t chunk_size;
uint64_t splitter_pos;
```

We can see these values getting initialized in the constructor below.

```
explicit WordsMR(char* _data, uint64_t length, uint64_t _chunk_size) :
         data(_data), data_size(length), chunk_size(_chunk_size),
         splitter_pos(0) {}
```

### 3.2.1 The locate() function

The first function declared in the public scope of the class is `locate()`:

```
1    void* locate(data_type* str, uint64_t len) const
2    {
3        return str->data;
4    }
```

The `locate()` function takes two parameters: a pointer to the input data type (`data_type*`), and a length (`len`). It returns a pointer to the start of readable data. In this case, it will be the data field of the `wc_string` struct, which is our input data type.

### 3.2.2 The map() function

The `map()` function is declared next. Its two parameters is the input data type (passed by reference), and the ouput data container (`out`). Remember, our input data type is `wc_string`. Remember that `wc_string` is a struct with two fields: `data` and `len`. The input represents a "chunk" of data which we want to parse words from and emit *(key,value)* pairs associating each word with the count of `1`:

```
1    void map(data_type const& s, map_container& out) const
2    {
3        for (uint64_t i = 0; i < s.len; i++)
4        {
5            s.data[i] = toupper(s.data[i]);
6        }
7
8        uint64_t i = 0;
9        while(i < s.len)
10       {
11           while(i < s.len && (s.data[i] < 'A' || s.data[i] > 'Z'))
12               i++;
13           uint64_t start = i;
14           while(i < s.len && ((s.data[i] >= 'A' && s.data[i] <= 'Z') || s.data[i] == '\''))
15               i++;
16           if(i > start)
17           {
18               s.data[i] = 0;
19               wc_word word = { s.data+start };
20               emit_intermediate(out, word, 1);
21           }
22       }
23   }
```

The first four lines of the function converts every character in the input chunk to uppercase. The function `toupper` is declared in the standard header file `cctype.h`. This is to ensure the word count application ignores case as it counts words.

The while code block contains the majority of the work . It contains two inner while loops and an if statement.

- The first inner while loop determines the starting point of the word. It skips any characters that are outside the ASCII range of A to Z. As soon as it exceeds the length of the string or hits an alphabetic character, it stops incrementing `i`. This value of `i` is designated as the start of the word (`start = i`).
- The second inner while loop determines the end of a word. It keeps incrementing `i`, so long as the current character is alphabetic, and the value of `i` is less than the length of the input chunk. The variable `i` stops incrementing once we hit a non-alphebetic character or the end of the input chunk.

The next few line of code is crucial to understanding how the word count applicaton works:

```
s.data[i] = 0;
wc_word word = { s.data+start };
emit_intermediate(out, word, 1);
```

Recall that `0` is the integer value of `\0`, the null terminator, which indicates where a string should be terminated. We define our key data (`wc_word word`) to be a pointer to our input chunk, at offset `start`. Next, we emit the word key with the value `1`.

The above process repeats until all the valid words in the input chunk are consumed.

### 3.2.3 The split() function

The `split()` function tokenizes the input prior to its going to the `map()` function:

```
1   /** wordcount split()
2    *  Memory map the file and divide file on a word border i.e. a space.
3    */
4   int split(wc_string& out)
5   {
6       /* End of data reached, return FALSE. */
7       if ((uint64_t)splitter_pos >= data_size)
8       {
9           return 0;
10      }
11
12      /* Determine the nominal end point. */
13      uint64_t end = std::min(splitter_pos + chunk_size, data_size);
14
15      /* Move end point to next word break */
16      while(end < data_size &&
17          data[end] != ' ' && data[end] != '\t' &&
18          data[end] != '\r' && data[end] != '\n')
19          end++;
20
21      /* Set the start of the next data. */
22      out.data = data + splitter_pos;
23      out.len = end - splitter_pos;
24
25      splitter_pos = end;
26
27      /* Return true since the out data is valid. */
28      return 1;
29  }
```

```
30
31      bool sort(keyval const& a, keyval const& b) const
32      {
33          return a.val < b.val || (a.val == b.val && strcmp(a.key.data, b.key.data) > 0);
34      }
35  };
```

Keep in mind while reading this code that the input file is in shared memory. The function takes a single parameter, a reference to a `wc_string` object, which is our input data type. Recall that the variables `splitter_pos` and `data_size` are global. The variable `splitter_pos` tells us where we are currently in the file. The variable `data_size` represents the size of the entire file. The variable `chunk_size` represents the size of each chunk.

The first if statement simply ensures that our current position does not exceed the bounds of the file. If so, we exit the function by returning 0 (indicating failure, and that there is nothing more to split).

We want each chunk to be approximately the same. We first determine a "nominal" end-point, or position to "chunk" the data:

```
uint64_t end = std::min(splitter_pos + chunk_size, data_size);
```

Obviously, this end-point won't always work. What if we land in the middle of a word? Therefore, we want to increment `end` until we hit a natural word boundry. The `split()` function declares this boundry as being either a space, tab, return carriage, or new line character. This is achieved by the following code:

```
while(end < data_size &&
          data[end] != ' ' && data[end] != '\t' &&
          data[end] != '\r' && data[end] != '\n')
          end++;
```

Once we determine a valid end-point, we populate the inputted `wc_string` object:

```
out.data = data + splitter_pos;
out.len = end - splitter_pos;
```

The starting point is set to data pointer plus the starting value of `splitter_pos`. The length is determined by subtracting `end` from `splitter_pos`.

Finally, we update `splitter_pos` to be the end point (`end`), and return `1` to indicate that we were able to successfully split the input.

## 3.3 The main() function

A simplified version of the `main()` function is shown below. This version only shows the non memory mapped code. The memory mapped version of the code can be viewed here. We also remove timing code for simplicity. The `CHECK_ERROR` function is defined in the file `stddefines.h` and is a useful wrapper for error handling.

```
1   int main(int argc, char *argv[])
2   {
3       int fd;
4       char * fdata;
5       unsigned int disp_num;
6       struct stat finfo;
7       char * fname, * disp_num_str;
8       struct timespec begin, end;
9
10      get_time (begin);
11
12      // Make sure a filename is specified
```

```
    if (argv[1] == NULL)
    {
        printf("USAGE: %s <filename> [Top # of results to display]\n", argv[0]);
        exit(1);
    }

    fname = argv[1];
    disp_num_str = argv[2];

    printf("Wordcount: Running...\n");

    // Read in the file
    CHECK_ERROR((fd = open(fname, O_RDONLY)) < 0);
    // Get the file info (for file length)
    CHECK_ERROR(fstat(fd, &finfo) < 0);

    uint64_t r = 0;

    fdata = (char *)malloc (finfo.st_size);
    CHECK_ERROR (fdata == NULL);
    while(r < (uint64_t)finfo.st_size)
        r += pread (fd, fdata + r, finfo.st_size, r);
    CHECK_ERROR (r != (uint64_t)finfo.st_size);


    // Get the number of results to display
    CHECK_ERROR((disp_num = (disp_num_str == NULL) ?
      DEFAULT_DISP_NUM : atoi(disp_num_str)) <= 0);

    get_time (end);

    #ifdef TIMING
    print_time("initialize", begin, end);
    #endif

    printf("Wordcount: Calling MapReduce Scheduler Wordcount\n");
    get_time (begin);
    std::vector<WordsMR::keyval> result;
    WordsMR mapReduce(fdata, finfo.st_size, 1024*1024);
    CHECK_ERROR( mapReduce.run(result) < 0);
    get_time (end);

    #ifdef TIMING
    print_time("library", begin, end);
    #endif
    printf("Wordcount: MapReduce Completed\n");

    get_time (begin);

    unsigned int dn = std::min(disp_num, (unsigned int)result.size());
    printf("\nWordcount: Results (TOP %d of %lu):\n", dn, result.size());
    uint64_t total = 0;
    for (size_t i = 0; i < dn; i++)
    {
        printf("%15s - %lu\n", result[result.size()-1-i].key.data, result[result.size()-1-i].val);
    }

    for(size_t i = 0; i < result.size(); i++)
```

```
    {
        total += result[i].val;
    }

    printf("Total: %lu\n", total);

    free (fdata);

    CHECK_ERROR(close(fd) < 0);

    get_time (end);

    #ifdef TIMING
    print_time("finalize", begin, end);
    #endif

    return 0;
}
```

We analyze this code in parts:

In the first part, we are simply setting up variables. Users running the program are required to input a file, and may choose to specify the top number of results to display. Variables are declared to facillitate file reading and command line parsing:

```
int fd;
char * fdata;
unsigned int disp_num;
struct stat finfo;
char * fname, * disp_num_str;
// Make sure a filename is specified
if (argv[1] == NULL)
 {
       printf("USAGE: %s  [Top # of results to display]\n", argv[0]);
        exit(1);
 }
fname = argv[1];
disp_num_str = argv[2];
```

We next open the file for reading, and get its size using the `fstat` function. We `malloc()` a block of memory, and have the descriptor `fdata` point to it. Next, we read the file into memory using the `pread()` system call. We also check to see if the user inputted the optional parameter that sets the maximum number of entries to display. If so, we update the variable `DEFAULT_DISP_NUM` to reflect this amount:

```
uint64_t r = 0;
fdata = (char *)malloc (finfo.st_size);
CHECK_ERROR (fdata == NULL);
while(r < (uint64_t)finfo.st_size)
       r += pread (fd, fdata + r, finfo.st_size, r);
CHECK_ERROR (r != (uint64_t)finfo.st_size);
// Get the number of results to display
CHECK_ERROR((disp_num = (disp_num_str == NULL) ?
             DEFAULT_DISP_NUM : atoi(disp_num_str)) <= 0);
```

Now the magic happens: we run our MapReduce job. This is easily accomplished in three lines. We first instantiate a `result` vector. We instantiate a mapreduce job with the line:

```
WordsMR mapReduce(fdata, finfo.st_size, 1024*1024);
```

Here, `fdata` will bind to the data pointer in `WordsMR`, `finfo.st_size` will bind to `data_size` and `chunk_size` wil be set to the quantity `1024*1024`. The following line just ensure the result array is non empty:

```
CHECK_ERROR( mapReduce.run(result) < 0);
```

The final part of the code prints out the top `DEFAULT_DISP_NUM` entries, sorted in order of greatest to least count. Since the output of the MapReduce task is in sorted descending order, it suffices just to print the first `DEFAULT_DISP_NUM` values. A second loop counts the total number of words found:

```
unsigned int dn = std::min(disp_num, (unsigned int)result.size());
printf("\nWordcount: Results (TOP %d of %lu):\n", dn, result.size());
uint64_t total = 0;
for (size_t i = 0; i < dn; i++)
{
     printf("%15s - %lu\n", result[result.size()-1-i].key.data, result[result.size()-1-i].val);
}
for (size_t i = 0; i < result.size(); i++)
{
        total += result[i].val;
}
printf("Total: %lu\n", total);
```

Finally, the `fdata` pointer is freed and we end the program:

```
free (fdata);
CHECK_ERROR(close(fd) < 0);
return 0;
```

## 3.4 Running the Code

We prepared a simplified version of the word count program, in this archive called phoenix++-wc.tar.gz, which shows what a standalone Phoenix++ application looks like. Alternatively, you can access the official Phoenix++ release at this link. The following instructions assume that you downloaded the phoenix++-wc.tar.gz file.

After downloading the file, untar it with the following command:

```
tar -xzvf phoenix++-wc.tar.gz
```

Let's look at this folder's directory structure:

```
|-- data
|   |-- dickens.txt
|   |-- sherlock.txt
|-- Defines.mk
|-- docs
|   |-- 2011.phoenixplus.mapreduce.pdf
|-- include
|   |-- atomic.h
|   |-- combiner.h
|   |-- container.h
|   |-- locality.h
|   |-- map_reduce.h
|   |-- processor.h
|   |-- scheduler.h
|   |-- stddefines.h
|   |-- synch.h
|   |-- task_queue.h
```

```
|   |-- thread_pool.h
|-- lib
|-- Makefile
|-- README
|-- src
|   |-- Makefile
|   |-- task_queue.cpp
|   |-- thread_pool.cpp
|-- word_count
    |-- Makefile
    |-- README
    |-- word_count.cpp
```

The folder `data` contains some sample data files for you to play with. The file `Defines.mk` contains many of the compiler flags and other directives needed to compile our code. The `docs` folder contains the Phoenix++ paper that you can read. The `include` folder contains all the header files we need for our Phoenix++ application. The `lib` directory is currently empty; once we compile our code, it will contain the phoenix++ library file, `libphoenix.a`. The `src` folder contains the code needed to make the Phoenix++ library file. Lastly, our word count application is located in the directory `word_count`.

**To compile the application, run the `make` command in the main Phoenix++-wc** directory:

```
make
```

Let's run the application on the file `dickens.txt`. This file is 21MB, and contains the collective works of Charles Dickens. Run the application with the following command:

```
time -p ./word_count/word_count data/dickens.txt
```

This will show you the top 10 most frequent words detected in `dickens.txt`. To see detailed timing information, uncomment the line `#define TIMING` in `include/stddefines.h`.

Below you will find a series of exercises to explore the example further. Happy analyzing!

## 3.5 Exercises

Let's explore the `word_count.cpp` file a bit futher by modifying it slightly. Remember, every time you change this file, you must recompile your code using the `make` command!

- Run the word count program to print the top 20 words, the top 50 words, and the top 100 words. How does the run-time change?
- Most of the words that have been showing up are common short words, such as "and", "the", "of", "in", and "a". Modify the `map()` function to only print out words that are five characters or longer. What are the top ten words now? How does Charles Dickens' top 10 words differ from Arthur Conan Doyle's?
- Use the setenv command to set the `MR_NUMTHREADS` environmental variable to a user inutted number of threads in `main.cpp`. Check the setenv documentation for more details.
- Check the number of CPU cores on your machine by checking `/proc/cpuinfo`. Vary the number of threads from *1...c* where *c* is the number of CPU cores. Plot your timing results using Matplotlib.
- *Challenge*: The words that are showing up are still those that largely reflect the grammar of an author's writing. These are known as function words. Modify the `map()` function to *exclude* any words that are function words. A list of 321 common function words can be found at this link.