Implement with Java Fork/Join Framework

Wei Huang(9524543)

1.Introduction

This report mainly introduces java Fork/Join for multicore systems. Fork/Join targets single node, multicore parallelism[3]. The fork/join framework make you take advantage of multi-processors, and it is an implementation of the ExecutorServiece interface. It is designed for task that can be divided into subtasks and then join them. (write some feature of forkjoinpool)

There already have some existed ThreadPool, such as ThreadPoolExecutor, and they can handle problems well in many times. But sometimes, traditional using threadpool may cause problems. Assuming we have a work that is running thread, when this work is broken into small two or more sub-work, the threadpool will initiate child-threads to execute, and it may initiate recursively. The bigger problem in that condition is that the parent threads need to wait for its child threads running out, so we need to apply some mechanism to ensure their concurrency. If we implement by ourselves, it may be combersome. The ForkJoinPool expects all tasks that are independent and can parallel run as possible. What's more, ForkJoinPool has another feature: work stealing. Each work thread has their own work queue which can be implemented by deque. When a task break a new thread, the new thread will be push into the head of deque. If one task executes the combination with another task which has not finished its work, it will push another task into the head of deque, rather than waiting for the combination task finishing(like Thread.join ()). Fork/Join would steal other tasks from the tail of another task's deque when its task queue is empty. If we use traditional ThreadPoolExecutor, we may be hard to use work stealing.

There are have many parallel programming model for large-scale machines, such as MapReduce, Phoenix, C+pthreads and so on. Fork/Join still has its advantages. We can cut the problem up into Fork/Join size pieces and have a thread pool work on that. Doing these in a random order requires much co-ordination at a hardware level. The overheads would be a killer. In Fork/Join, tasks are put onto a queue that the thread reads off in Last In First Out order (LIFO/stack), and work stealing (in core work, generally) is done First In First Out (FIFO/"queue"). The result is that long array processing can be done largely sequentially, even though it is broken into tiny chunks. Fork/Join allow more efficient use of hardware threads in an uneven situations.

In the next section, I will introduce the relevant literature, and explain and summary their main opinion. Section 3 describe the simple benchmarks and their implementation in Java Fork. In section 4, the paper will discuss the difference between parallel programming models.

2.Relevant Literature

Evaluating MapReduce for Multi-core and Multiprocessor Systems

MapReduce was designed by Google to process big data over thousands of servers. Phoenix is an implementation of MapReduce for shared-memory systems. "The Phoenix run-time automatically managers thread creation, dynamic task scheduling, data partitioning, and fault tolerance across processor nodes". The scheduler in the Phoenix Runtime System controls the implementation of runtime, and creates, manages all Map and Reduce threads. In order to initialize the Map phase, the scheduler invokes the Splitter to break the input into the same size units for Map processing. The Map tasks are allocated dynamically to workers and then emit the intermediate value pairs <Key, value>. The Partition function divides the emitted pair values with the same key into Reduce tasks which also allocated to worker dynamically. The Reduce may cause large load unbalance. The Map-Reduces buffers are assigned to store the emitted pair value after Map phase, while the emit-intermediate function store all <key,value> pairs with the same key to the same buffer. Phoenix employs all scheduling methods to to the below four conditions: (1) Assigning workers to all available cores. (2) To achieve load balance, Phoenix spawn Map and Reduce tasks to workers dynamically. (3) Each Map function process one unit each iteration. If the assigned unit is over one, Map deal with the next one after the previous one finishing. Phoenix adjust the unit size to can fit in the L1 cache. (4) The Partition function determine how to handle the intermediate values.

This paper also use the eight benchmarks to compare Phoenix in CMP with Phoenix in SMP. **Figure 1** shows the Speedup with Phoenix for the large data sets as we scale the number of processors cores in the CMP and SMP. **Figure 2** presents the different time for the CMP system. **Figure 3** presents CMP speedup with 8 cores as we vary the data set size. This paper shows that Phoenix generates higher performance both in CMP and SMP architectures, so MapReduce is a good architectures for parallel in shared-memory systems. **Figure 4** compare the performance between Phoenix with P-Threads in CMP and SMP systems.

Phoenix:Modular MapReduce for Shared-Memory Systems

This paper demonstrates the rewriting of Phoenix, a MapReduce framework for shared-memory CMPs and SMPs[2]. In the Phoenix++, it provides two abstractions: containers, which supports group-by the intermediate values and stores them in combiners between the map and reduce phases, and combiner objects store the emitted values with the same key which is similar to reduce. That paper shows three containers: hash container, array container, and common array container. User can implement their own particular functions interface through the common interface: initialize(num_map_threads, num_reduce_tasks), container get(), put(container), and combiner_iterator out(reduce_task_id). That paper demonstrates that ”In order to maximally reduce memory pressure due to intermediate key-value storage, Phoenix++ invokes the combiner immediately after each key-value pair is emitted by the map function”. There are also many interfaces: initialize(), add(value), add(container), discard(), and value* next() designed for use to implement their own functions. Next, the authors give many experimental results to compare the performance between Phoenix++ and Phoenix 2. For total performance, the Phoenix++ is consuming less time and more scalable than Phoenix 2 with all benchmarks which can be seen in **Figure 5**. Finally , the paper draw a conclusion that “shared-memory MapReduce frameworks can achieve much higher performance by adapting to the characteristics of their workloads”.

Comparing Fork/Join and MapReduce

According to this paper, “Java Fork/Join has low startup latency and small inputs (<5MB), but it cannot process larger inputs because of the multicore server’s limitation”. Hadoop process input data in cluster, distributed, and parallelism, so it have high startup latency and better performan- ce in bigger input data. The fork function create a new task that can parallelism run with all tasks. The join function merge the result when the child and parent tasks have all finished. All tasks are put in ForkJoinPool which can call the submit, execute, and invoke to perform the given task. The dynamic work load is efficiency, because the running time is not only determined by the number of work of task, but also the value in the input data sets. The Fork/Join allow inter-task communication, which the tasks are independent in MapReduce.

The paper discuss the performance between Fork/Join and MapReduce with file of size 0.003 MB, 0.06MB,1.49MB,4.79MB, and 104.3MB. The Fork/Join program fails with an OutOfMemory when the size of file is 104.3MB.

3.Desciption of Implementation

Matrix Multiplication: We use ForkJoin Framework, create a ForkJoin task, which provides the mechanism to execute the operations of fork() and join(). In Matrix Multiplication, it extends RecursiveTask, and return the double[][] results.

In the override of compute(), we first compare the length of task with the value of Threshold, or divide the task into the number of threads tasks, and call the function of fork to process the subtask, and join() function merge the subtasks’ results(the matrix is generated by random method).

```
@Override
        protected double[][] compute() {
                List<MatrixMultiply> tasks = new ArrayList<MatrixMultiply>();
                int row_num = (end - location)/numthread;
                if((end - location) <= Threshold){
```

```
                System.out.println("fork:"+this.getPool().getPoolSize());
                return Fork_Multiply();
        }else{
                for(int i = 0 ; i<numthread; i++){
                        MatrixMultiply upper_matrix = new MatrixMultiply(matrix_A,matrix_B,(location+row_num*i),
                                        (location+row_num*(i+1)),matrix_len,Threshold,numthread,output);
                        upper_matrix.fork();
                        tasks.add(upper_matrix);
                }
                for(int i = 0; i<tasks.size();i++){
                        tasks.get(i).join();
                }
                return output;
        }
```

the main implementation of Fork_Multiply()

```
while(temp_loca < end){
                b_ptr = 0;
                for(int i = 0; i< matrix_len; i++){
                        for(int j = 0; j< matrix_len; j++){
                                output[temp_loca][j] += matrix_A[temp_loca][i]*matrix_B[b_ptr][j];
                        }
                        b_ptr++;
                }
                temp_loca++;
        }
        return output;
```

In the main function, ForkJoinPool is constituted with ForkJoinTask array and ForkJoinWorkerThread array. ForkJoinTask array is responsible for storing the tasks that the program submits to ForkJoinPool, and ForkJoinWorkerThread array processes this tasks.

```
ForkJoinTask<double[][]> matrixTask = new MatrixMultiply(matrix_A,matrix_B,0,matrix_len, matrix_len,Threshold,
numthread,output);
ForkJoinPool fjpool = new ForkJoinPool();
Future<double[][]> result = fjpool.submit(matrixTask);
```

The principle of ForkJoinTask's fork is when we call ForkJoinTask's fork(), the program will revoke pushTask of ForkJoinWorkerThread to asynchronously execute this task, and then return the results immediately. The function of pushTask is putting the current task in the queue of ForkJoinTask array, and then call the signalWork() of ForkJoinPool to awark or create a work thread. Join function is mainly block the current thread and wait for the results. At first, it invokes doJoin(), and through doJoin() get the state of current task to judge what results should return. The states of task have four kinds: NORMAL, CANCELLED, SINGAL, AND EXCEPTIONAL.

Histogram: This program is to compute the frequency of the value between 0-255 range of RGB pixels. It assigns points to different tasks, which compute the frequency and insert it into the array tempcolor[]. In the override of compute() function, the implementation of division is similar with the above Matrix Multiplication(so I do not display this part of code), and just the specific processing of the task is difference when the length of it is up to the value of Threshold9(this program need to input the image filepath).

```
for(int i = start; i<= end ; i += 3 ){

                        tempcolor[pixels[i] & 0x000000ff] ++;
                        if((i+1)< 144000)
                        tempcolor[pixels[i+1]& 0x000000ff + 256] ++;
                        if((i+2)< 144000)
                        tempcolor[pixels[i+2]& 0x000000ff + 512] ++;
                }
                return tempcolor;
```

The rest four implementation can be seen from the code, because it is too long to put in here. Kmeans need to be computed cyclically until each cluster is the same with the previous loop.

Kmeans: It divides the a set of input data into clusters. Because it is iterative, the ForkJoinTask is called many times until the condition is satisfied. This program create HashMap<Double[],Vector <Double<>>>, and the parameter of Double[] is the mean point of each cluster, and the Vector<Double<>> store the whole points that are belong to this mean point. Calling the method of Join wait and return the results, and combine different tasks' results into one HashMap: termi_map, which is the final result.

Linear Regression: It computes the best line that fit a set of input data. In the program, it just need to put the sum of all X*X, Y*Y, X*Y, Y, and X into the final return map.

String_Match: String Match application scrolls through a list of keys (provided in a file) in order to determine if any of them occur in a list of encrypted words (which are hardcoded into the application). Each task parses its input words and returns a word in the "keys" file as the key and a flag to indicate whether it was a match as the value.

Word_Count: It computes the frequency of each word in the input file. The subtasks processes its own parts and generates its intermediate results through Join() method.

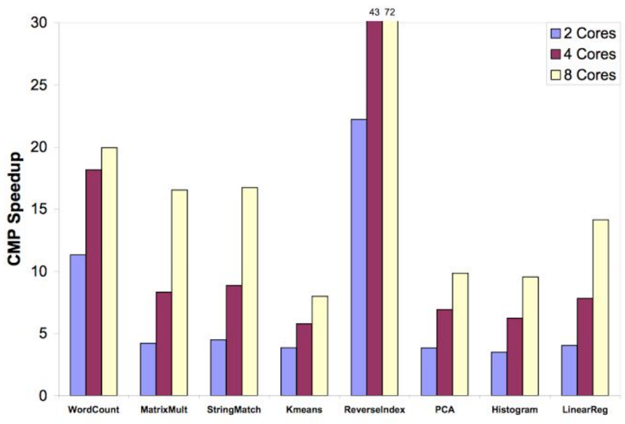
PCA: The PCA application computes the mean vector and covariance matrix for a randomly generated matrix, which is the first step in performing principal component analysis. It includes two class, the first class is to compute the mean vector, and then the mean vector as input parameter to compute covariance matrix.

4.Performance Results

The speedup of these eight applications can be seen from the below **figure 6**. With the threads from 1, 2, 4, 8 and 16, eight applications have different speedup. For the big data size of Matrix_Multiply(500 and 1000), the speedup increase with the number of threads from 1 to 16, while if the size of input data is small, the speedup would decrease which can be seen from Matrix_Multiply(100) of the figure6. The result for this condition maybe that if each task's size is too, the division time and waiting for results time are overhead. For the application of KMeans, when the number of threads is over 8, the performance decreases, because the KMeans application need iterative, the next ForkJoinTask must wait for the previous ForkJoinTask finishing. According to the figure6, we can get that the main consuming-time factors are consuqer-time, join() waiting time. It is obvious that for PCA application when we choose the size of matrix row and column in 1000X 1000, the speedup is up to perfect, so we can get that for the big size of data sets, the waiting time in the join() phase can be ignored compared with the computing time.

The difference performance of ForkJoin, Phoenix, and Pthreads can be seen from the below figures. In figure 4, Phoenix and PThreads are good frameworks for some classes of computation. For two applications:Wordcount, Matrix_mult, Phoenix is better performance than PThreads both in CMP and in SMP. For three applications: KMeans, PCA, and Histogram, PThreads outperformance Phoenix significantly. Phoenix is not good enough to cover all applications. Phoenix is more similar with ForkJoin, which is need to dynamically conseque the tasks into several small subtasks, and then merges the intermediate results, and in the phase of the middle of fork() and join(), it must wait for the results. Before the beginning of reduce(), it also must wait for the intermediate results that the map() generates. The each thread unit's size for Phoenix and ForkJoin can decided by the input data sets. "The P-threads code uses the lower level API directly and has been manually optimized to be as fast as possible"[1]. According to the figure 5, the Phoenix++ has the best speedup, because the MapReduce performance is mainly limited by disk and I/O, it use shared-memory can reduce this obstacles.

Figure1 Speedup for Phoenix when we vary the number of cores



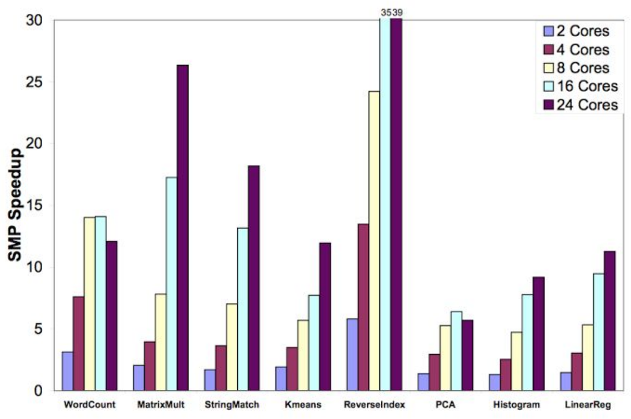


Figure2 The different parts’ time of CMP system

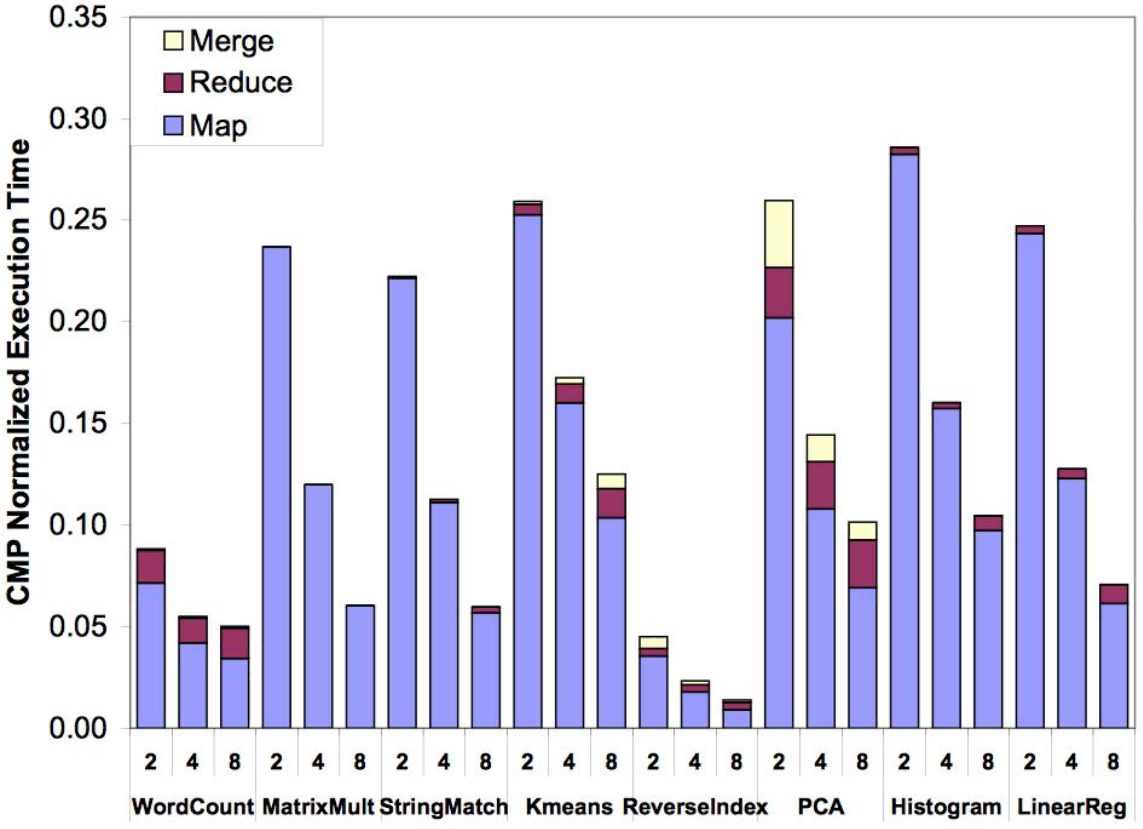


Figure3 The speedup for Phoenix when we vary the size of data set.

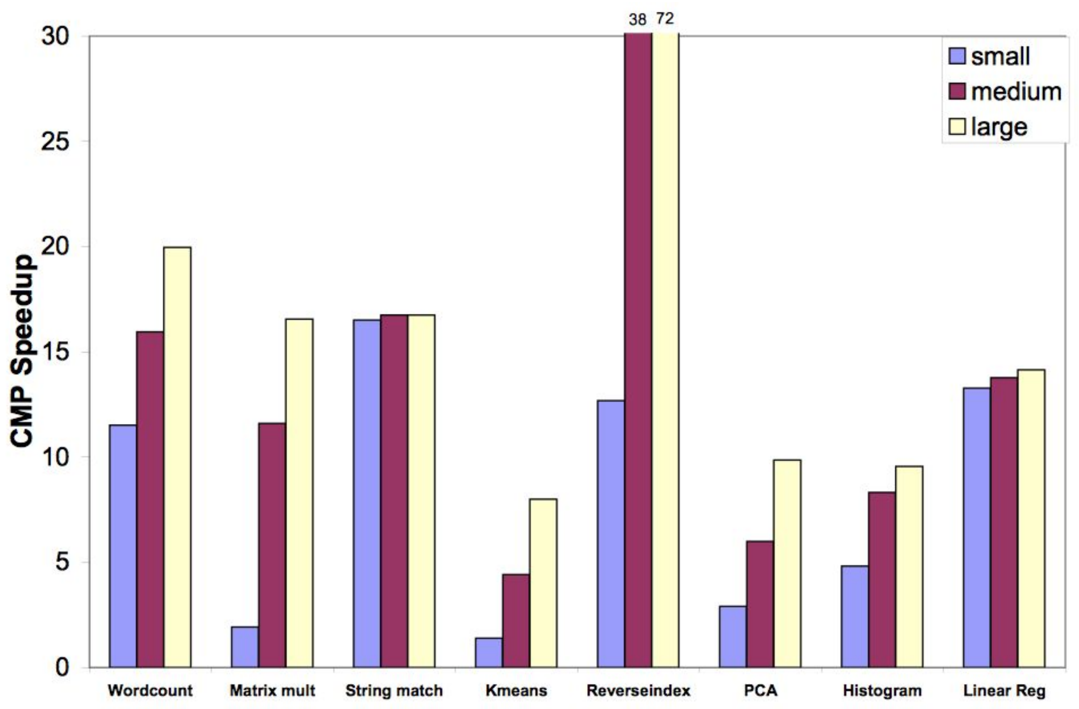
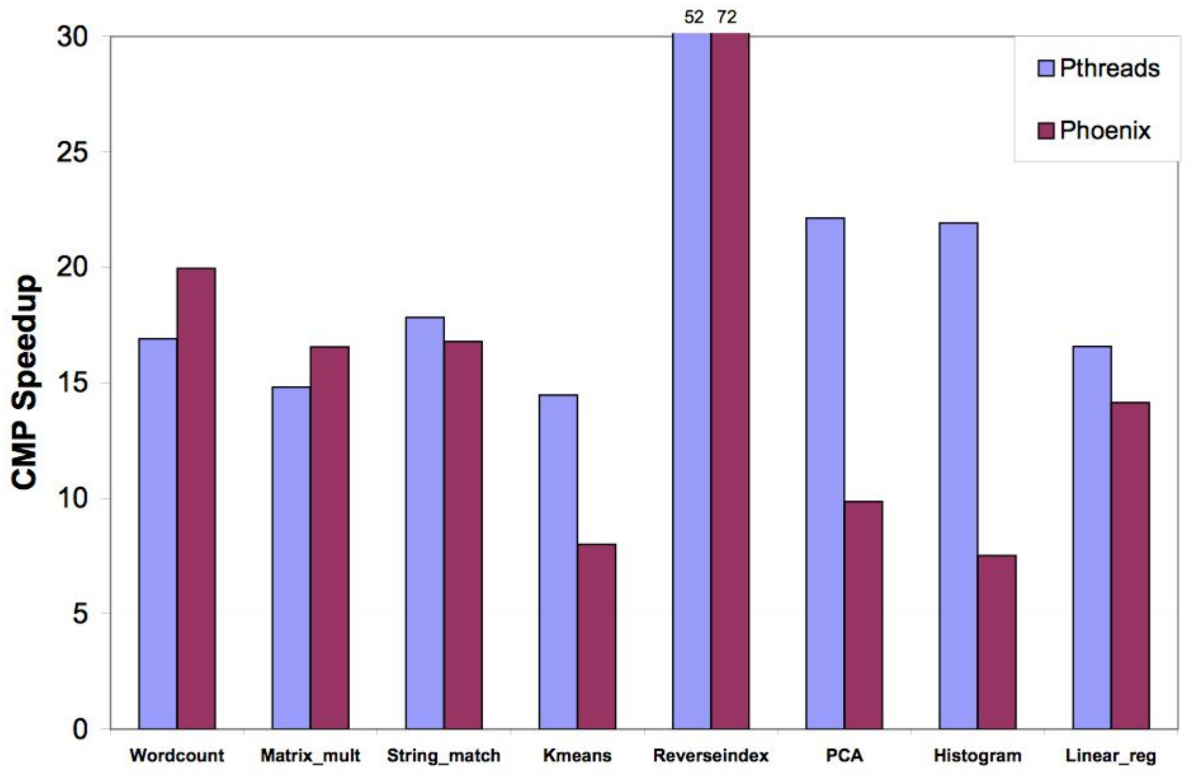


Figure4 The comparison between Phoenix with P-Threads in CMP and SMP systems



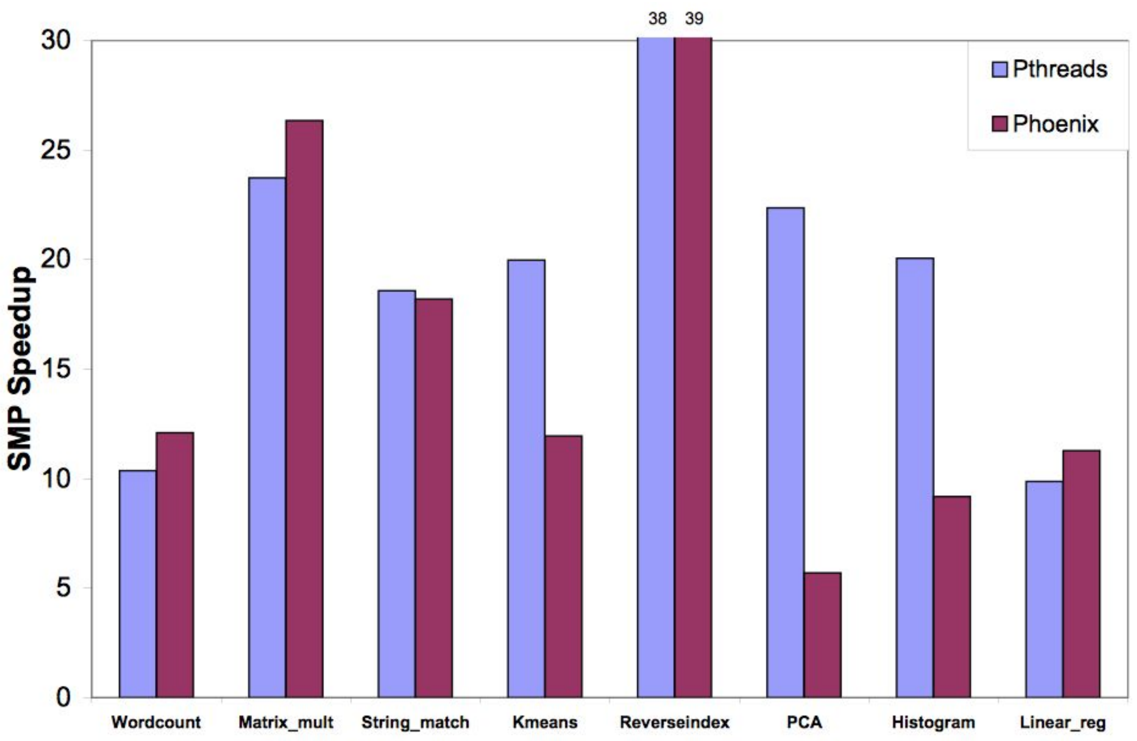


Figure5 The top line is speedup, and the bottom line is scalability comparison of Phoenix++ (blue) and Phoenix 2(orange).

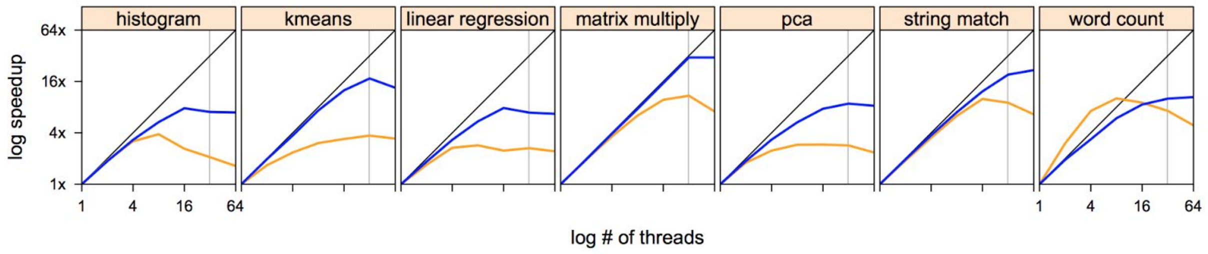
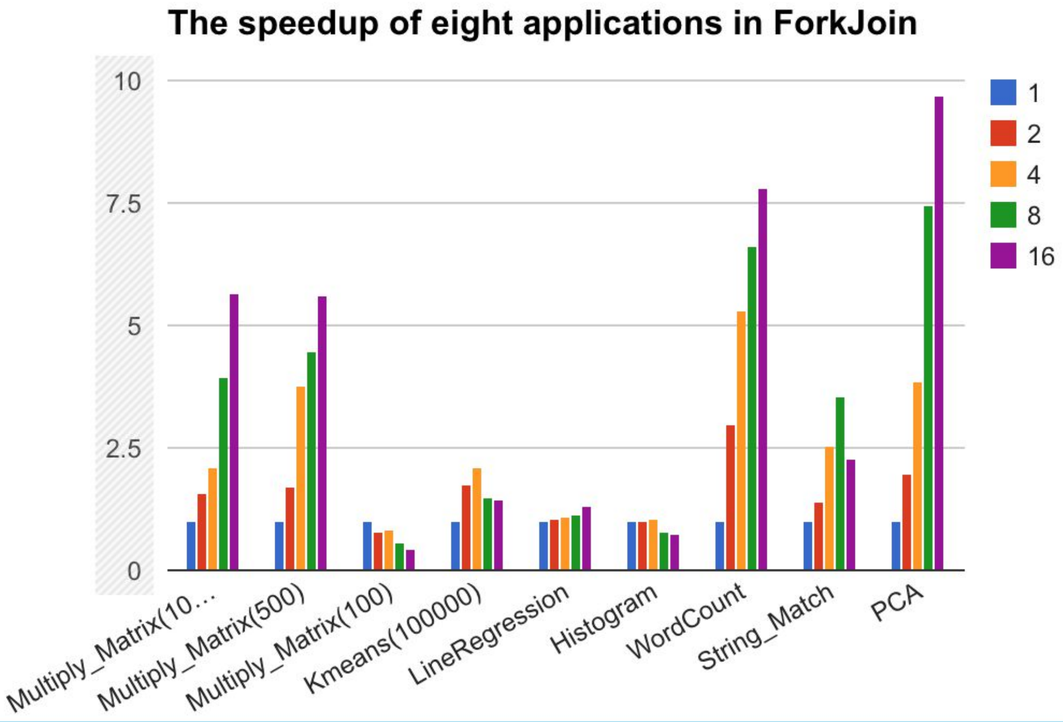


Figure6 the speedup of each application in ForkJoin FrameWork



5.Summary

This report describes my designing applications in Java Fork/Join Framework, and explains the principle of Fork/Join framework. Next, this report introduces some relevant papers which is related with MapReduce, P-threads, Phoenix++, and Fork/Join frameworks. The section 3 displays some mean codes and implementation methods for the eight applications, because the main part code for these eight applications is similar, so I just write some main implementation methods. In the final part, I compare different frameworks' performance and analyze the reasons.

References:

1. Evauating MapReduce for Multi-core and Multiprocessor Systems
2. Phoenix++: Modular MapReduce for Shared-Memory Systems
3. Comparing Fork/Join and MapReduce