Feature Extraction Algorithm Optimization for Multicore Architecture in Java

Tianshuo Deng Maochen Guan
Computer Science Department
New York University
{td859,mg3364}@nyu.edu

Abstract

In this paper, we investigate different methods to optimize performance in multicore architecture for feature extraction procedure in TweetEmotion.com. And feature extraction from sentences is one of the time consuming procedure. This paper focus on reducing training time and response time by parallelizing training procedure and parsing algorithm. Furthermore, we investigate bottom up parsing technique and demonstrate how to produce a non-locking algorithm.

1 Introduction

TweetEmotion.com provides sentiment analysis service for tweets using natural language processing techniques. Given a tweet and a topic, the algorithm used in TweetEmotion.com classifies the tweet to 3 classes, which are Positive, Negative and Neutral, according to that topic. For example, given "I love iPad" and topic "iPad", the tweet will be classified as positive. Machine learning algorithm, Maximum Entropy, is used to do the classification.

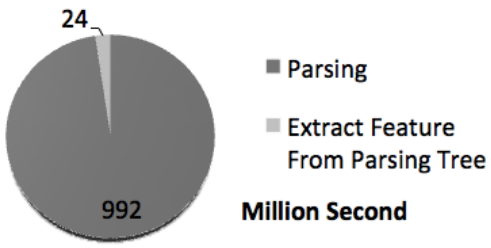
This paper discusses the optimization of feature extraction procedure for TweetEmotion.com to reduce both **training time** and **response time**. The techniques used for parallelization in this paper can be applied to NLP(Natural Language Processing) applications widely.

We first introduce the support for concurrent programming in latest Java release and the programming models including ThreadPool and Fork and Join Model. Then we discuss two scenarios where the feature extraction procedure can be optimized.

Parallel Training In training scenario, massive data are divided into batches and processed concurrently. Parallelization of training procedure reduces the time for training models in NLP system.

Parallel Parsing In execution scenario, parsing takes most of the time during feature extraction (Figure[1]). in order to improve the response time for an NLP system, parsing time for each sentence needs to be optimized. In Section[5.4], context-free parsing algorithm CKY is analyzed and parallelized. This paper investigates different ways of optimizing the CKY algorithm and compare those different approaches. The techniques, lock-free algorithm, bottom-up approach can also be applied to other dynamic programming algorithms.

Figure 1: Time of extracting feature from a sentence of length 20



2 Literature Survey

2.1 Concurrent Programming in Java

Divide-and-Conquer Fork and Join technique for parallelization is widely used in scientific computing. After dividing a problem into two or more sub-problems, the method solves the sub-problems in parallel. Typically, the subproblems are solved recursively and thus the next divide step yields even more sub-problems for solving in parallel.

In paper[**?**], Fork and Join framework is implemented using Thread class in Java. It also discusses the heart of Fork and Join framework the stealing scheduler. The heart of a Fork and Join framework lies in its lightweight scheduling mechanics.

Nonetheless, it doesn't investigate the latest feature provided by Java 7. In our project, we will examine how to utilize the latest native support for Fork and Join model .

In paper[**?**], it proposed how to implement work stealing. It allows efficient parallelization of irregular workloads. In our feature extraction process, workload varies depends on the complexity of each tweet. Those research focuses on the implementation of Fork and Join without investigating the runtime impact for the Fork and Join model. They are not focused on the domain of NLP. In our project we will evaluate the usage of Fork and Join Model and compare it with a simple Thread Pool mode. The runtime impact under Java 7 is experimented with different memory size, thread number.

2.2 Parallelization in Natural Language Processing

In NLP with machine learning approach, map-reduce is widely used. But generally, map-reduce is used for multiple machines with distributed memory. This paper[**?**] describes ways to do distributed computing for Natural Language Processing, which divides training data among multiple CPUs for faster processing.

However, this paper does not discuss how to utilize multicore architecture. In our project, not only do we split training data into different batch/tasks, but also for each training record, we are taking advantage of shared memory, which means we have threads of parsing algorithm working in parallel for the same training record that allows us to do parallelization in a more fine-grained level and improve the response time of a NLP system.

3 Proposed Idea

First, the sequential program of feature extraction is modified to be thread-safe to support multi-threading.

Then we parallelize the feature extraction procedure for training on data level. Data are partitioned into batches, which reduces the training time of the system. Fork and Join model and Thread Pool model are applied and compared.

Beyond that, the parsing algorithm used in the procedure can also be parallelized. Most parsing algorithms use dynamic programming. In CKY algorithm, both bottom-up and top-down approaches can be used. This paper will discuss how to use a bottom-up approach to implement CKY algorithm where most threads joining/blocking can be avoided when using non-blocking algorithm and avoiding shared data among threads.

Finally we compare those approaches and see how they perform in different software and hardware configurations. As later discussion, when memory is sufficient, by maintaining state for each threads, performance can be greatly improved comparing to using shared data among threads. But when memory is limited, it may introduce frequent GC operations which degrades the performance of the system.

In practices, decisions should be made based on the environment and setup of the NLP system.

4 Experimental Setup

Language We use Java 7 as language of choice. In the area of Natural Language Processing, Java is widely used, because of its sufficient support to regular expression and text processing. The Object-Oriented structure also benefits the work in software engineering. JVM is highly optimized and platform independent. Nowadays more and more applications are developed on JVM, so the result of our project will be widely applicable for NLP applications and applications running on JVM platform.

Parsing Framework CKY is used to parse the input and construct the parsing tree. It is a dynamic programing method for parsing a sentence with context-free grammar.

Profiling YourKit is a famous Java profiling tool, it supports multi-threaded profiling. We use YourKit to profile following aspects of our program in different parallel model:

1. Execution Time

   In feature extraction procedure, there is no interaction with other procedures. It could be considered as a single step in text mining. So execution time is an important metric. In the experiment, we measure the time for extracting features from a certain amount of tweets.

2. CPU usage

   We compare the cpu usage and its distribution on multicore architecture.

3. Thread Status

   From the profiler, we check the status of threads and will see if a specific thread is blocked, sleeping, waiting or running.

4. Memory Usage

   Memory usage module identifies whether the bottleneck is CPU or memory.

5 Experiments and Discussion

5.1 Feature Extraction Procedure

Feature extraction procedure is executed when training the classifier and getting new input from user. In TweetEmotion.com the procedure is defined as: Given the input sentence, output the features for that sentence. There are two issues that hinder the improvement of performance in most modern NLP application.

First, the data size involved in training a model is huge. This is a very common issue for statistical machine learning algorithm. With the increment of data size, training time for a NLP system is significantly increased.

Second issue is specific for Natural Language Processing application. Parsing a sentence is slow. In NLP applications, parsing is an essential step for in-depth analysis. It produces a parsing tree for a sentence which reveals phrase structures and their dependencies. Two widely used parsing algorithms, CKY algorithm and agenda parsing algorithm, take exponential time relative to the length of the sentence. Longer parsing time means longer response time for an NLP system.

In Section[5.3], we discuss the first issue by parallel training data. In Section[5.4] we discuss how to optimize the parsing algorithm for parallelization.

5.2 Concurrent Programming in Java 7

5.2.1 Thread Pool Model

Thread pool model is a concurrent programming solution provided by Java natively. It reduces the cost of creating new thread per request. Since Java 7, several classes are added to better support thread pool programming model. In addition, user can designate the number of threads which is an important factor affecting the performance.

5.2.2 Fork and Join Model

In Java 7, support for fork and join model is added. By implementing RecursiveAction or RecursiveTask interface one can easily define and implement a fork and join model. Similar to thread pool model, fork and join model also uses thread pooling where parallelism can be specified. By default it uses the number of cores as the number of active threads in the thread pool. Moreover, work stealing is implemented to solve the imbalanced workload issue.

5.2.3 Concurrent Data Structures

Barrier In this project, we prevent joining the threads spawned in order to get rid of threads blocking. Nonetheless, main thread still needs to be coordinated with the working threads. Thus, a barrier is used for this purpose. The barrier concept is the same as OpenMP barrier. In Java, a class called CyclicBarrier() is defined for the barrier type. And barrier.await() method can be invoked at the end of every thread.

AtomicInteger In bottom up parsing which is discussed in Section[5.5], to build a lock-free algorithm, dependency counter is used to track dependencies among cells. Thus, every cell should maintain a dependency counter to represent its state. However, the dependency counter will be manipulated by multiple dependent cell threads. Hence, by using AtomicInterger as dependency counter, the race condition can be avoided.

ConcurrentHashmap In Section[5.6], ConcurrentHashmap is used to store the best parse for each cell. ConcurrentHashmap allows more fine-grained locking comparing to Hashtable. Locking is added for writing to the same key in the ConcurrentHashmap. It provides thread-safety while not sacrificing performance too much.

5.3 Training Data Level Parallelization

Extracting features from training data is an embarrassingly parallel process because there is no dependency between data. It means the feature extraction procedure can be break down to batches and executed concurrently.

5.3.1 Fork and Join model

A problem is break down to subproblems of smaller size and solved recursively. Following code shows how to recursively extract features from a huge dataset:

Listing 1: fork dataset

```
protected  List<List<String>> compute() {
   if(high - low <= SEQUENTIAL_THRESHOLD) {
       return atomicCompute();
   } else {
       int mid = low + (high - low) / 2;
       ExtractionTask left  = new ExtractionTask(
           sentenceSet, low, mid);
       ExtractionTask right = new ExtractionTask(
           sentenceSet, mid, high);
       left.fork();
       right.fork();
       List<List<String>> rightAns = right.join();
       List<List<String>> leftAns  = left.compute();
       rightAns.addAll(leftAns);
       return rightAns;
   }
}
```

The base case of the fork and join task is parsing and extracting features from the given sentence directly.

5.3.2 Thread Pool Model

In the thread pool model, we split input data into batches and assign feature extraction threads based on the batch and join all of the result by the future returned by children threads. The following is batch division code

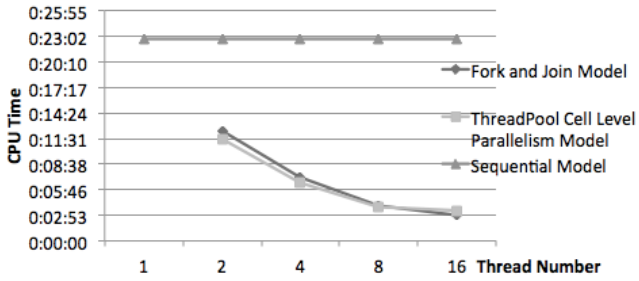
Listing 2: batch process using thread pool

```
for (int i = initIndex; i < tweetList.size(); i = i
     + batchNumber) {
    int startIndex = ((i - batchNumber + 1) > 0) ? (
        i - batchNumber + 1) : 0;
    List<TweetWithFilename> batchEntries = tweetList
        .subList(startIndex, i + 1);
    Callable<List<String>> extractionThread = new
        ExtractionCallable(batchEntries);
    Future<List<String>> submit = executor.submit(
        extractionThread);
    futureList.add(submit);
}
```

5.3.3 Result

We benchmarked those two models on 86 sentences from Penn Treebank on 16-Core 2.1GHz AMD Opteron 6272 machine with 256GB memory. From Figure[2], training time is greatly reduced in parallel programs. But there is little difference between fork and join model, and thread pool model. Because unlike function used in scientific computing, there is less joining operations in the fork and join model when parallelizing training. Two models essentially performs same operations.

Figure 2: Parsing time for 86 sentences



5.4 CKY Parsing Algorithm

The CockeYoungerKasami (CYK) algorithm (alternatively called CKY) is a parsing algorithm for context-free grammars. It employs bottom-up parsing and dynamic programming.

The importance of the CYK algorithm stems from its high efficiency in certain situations. Using Landau symbols, the worst case running time of CYK is $\theta(n^3 * |g|)$, where n is the length of the parsed string and $|G|$ is the size of the CNF grammar G. This makes it one of the most

efficient parsing algorithms in terms of worst-case asymptotic complexity, although other algorithms exist with better average running time in many practical scenarios.

The following is the brief step of the CKY Parsing:

1. The CKY Parsing creates a triangular table representing all spans in the sentence from 0 (the position before the first word) to N the position after sentence of length N.
2. Traverse for each possible combination

Listing 3: traverse pairs

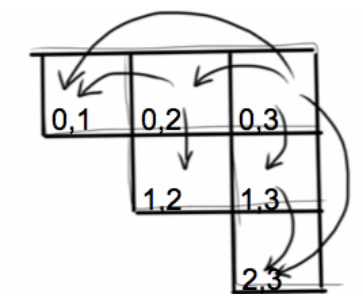
```
For j from 1 to N do:
    Fill in one span of length 1 using a POS
        rules, e.g., V     ate
      For i from 0 to j-2
        For k from i+1 to j-1:
          Add all matching nonterminals to [i,j]
              in table
```

5.5 Parallel Parsing on Cell Level

5.5.1 Pruning dependency

As discussed in Section[5.4], there are dependencies between two cells for different spans. For example, to calculate the best parse from $span(0,3)$, the best parse for $span(0,1), span(1,3)$ and $span(0,2), span(2,3)$ should already be generated. Figure[3] illustrates the dependency between cells for each span.

Figure 3: Dependency of Cells

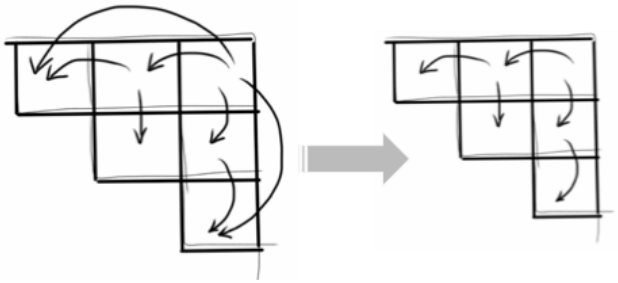


Each arc represents dependency between two cells. From the above graph there are redundant dependencies that are transitive. For example, if $cell_{0,3}$ depends on $cell_{0,2}$, and $cell_{0,2}$ depends on $cell_{0,1}$, then we can conclude that $cell_{0,3}$ is dependent on $cell_{0,1}$ without explicitly specifying the dependency between $cell_{0,3}$ and $cell_{0,1}$. This kind of redundant dependency can be pruned to reduce the number of coordination among cell threads. Figure[4] illustrates a CKY chart with pruned dependency.

5.5.2 Bottom Up Approach (Non-blocking algorithm)

CKY algorithm can be implemented in both bottom-up and top-down approaches. In bottom-up approach non-blocking algorithm can be implemented to eliminate threads joining or blocking. The algorithm is defined as the following:

Figure 4: Pruning Dependencies



1. Set dependency count to 2 for all cells except the diagonal. (As discussed in Section[5.5.1], after pruning, each cell only depends on its beneath and left cell)
2. Compute the current cell
   (a) For the diagonal cells, filling the cell using lexicon tagger. (For example, for the word drive, it may have probability of 65% of being a verb and 35% of being a noun)
   (b) For the rest, filling the cell by applying grammar: $cell_{i,j} \longrightarrow cell_{i,k} \;\; cell_{k,j}$
3. Decrease the dependency count for the above and right cell by 1. If the dependency count is decreased to 0, spawn the cell thread for that cell:
   $$dependencyCount(cell_{i-1,j}) - 1$$
   $$dependencyCount(cell_{i,j+1}) - 1$$
4. Repeat step 2 until $cell_{0,length_{sentence}}$ is computed

In bottom-up approach, most thread joining or blocking steps in top-down parsing can be avoided. Threads are spawned only when its dependencies are satisfied. Hence, there are less chance for threads blocking and waiting.

We use atomic integer to track dependency to avoid race condition when multiple cell threads accessing the same cell. Barrier is used to coordinate the cell threads and the main thread. Barrier.await() method will be called only after the parsing is finished.

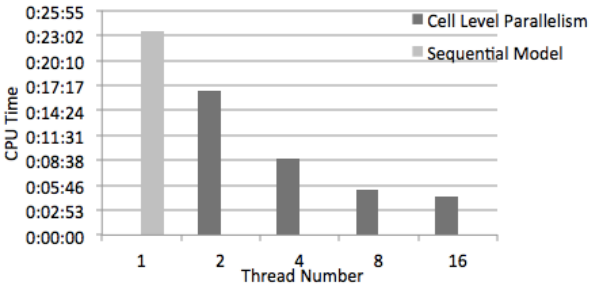
Listing 4: Barrier coordinates main threads and cell threads

```
if (cell.i == 0 && cell.j == cell.CKYTable.
    sentenceLength)
              parser.barrier.await();
```

5.5.3 Result

The program is tested on 16-Core 2.1GHz AMD Opteron 6272 machine with 256GB memory. As shown in Figure[5], parsing time for 86 test sentences is greatly reduce comparing to sequential program. With more threads, the parallel algorithm take the advantages of multicore architecture.

Figure 5: Parallel Non-locking CKY parsing compared with Sequential CKY parsing



5.6 Pair Level Parallelization

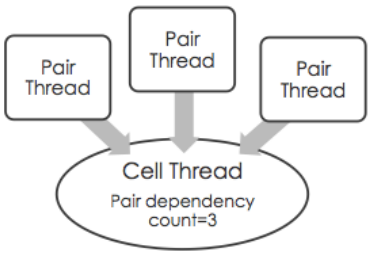
From Section[5.5], the workload is imbalanced in the cell level parallelism. With the processing of the cells, parallelism is limited by the length of the diagonal. And once proceeded to the last cell, only single thread is running with huge workload. It has to calculate the pairs, number of which equals to the $Length_{sentence}$. Thus, in order to distribute the workload for cell threads, pair level parallelization is introduced. Within a cell thread, pair threads are being assigned to calculate each pair of spans. Hence, even in the last cell, $Length_{sentence}$ threads will be spawned.

5.6.1 Coordination between cell threads and pair threads

As mentioned above, there are dependencies between cells, specifically, $cell[i][j]$ depends on $cell[i-1][j]$ and $cell[i][j+1]$. Thus, pair dependency counter (Figure[6]) is introduced to track the dependencies. By updating the counter, cell thread can be spawned. The following shows the coordination between cell thread and pair threads using pair dependency counter to avoid locking:

1. decrease the dependency counter in $cell[i-1][j]$ and $cell[i][j+1]$ by 1.
2. spawn the cell thread of $cell[i-1][j]$ (above cell) if the dependency count value is 0.
3. spawn the cell thread of $cell[i][j+1]$ (right cell) if the dependency count value is 0.

Figure 6: pair dependency count



5.6.2 Coordination among pairs

In this type of coordination, pair threads need to update the cell that contains the possible parsing result. Thus, there are two different approaches.

Sharing state among pair threads requires a synchronized shared stateMap object in cell thread. All subsided pair threads update the same stateMap object. In this project, stateMap object is implemented by ConcurrentHashMap that discussed in Section[5.2.3].

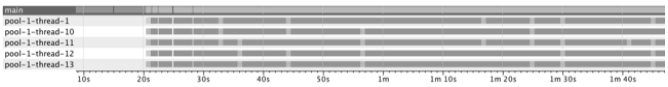
This approach doesn't require extra result merging step in cell threads. Nonetheless, it introduces lock (Figure[7]) because shared stateMap object has to be synchronized. As a result, it may cause potential pair threads blocking that decreases the performance.

Figure 7: Pair threads blocked

Separate state for each pair thread , on the other side, allows every pair thread have its own stateMap object. After pair threads finish, cell thread needs to merge all of the stateMaps holding by the pair threads. Its similar to MapReduce model used in distributed computing.

The advantage is that no coordination is required among pair threads. Since pair threads do not write to the same data structure, no data synchronization is required (Figure[8]).

Figure 8: Pair threads without blocking



However, maintaining states within each thread requires significant memory space.

5.6.3 Result

In the experiment, there are 39832 training sentences and 86 test sentences from Penn Treebank. When memory is sufficient, separate state for each pair thread increases the performance by eliminating synchronizations among shared data (Figure[9]). But on the other side (Figure[10]), it takes much more memory and in an computing environment where memory is limited, it may degrade the performance due to frequent garbage collection operations.

Figure 9: CPU Time Comparison

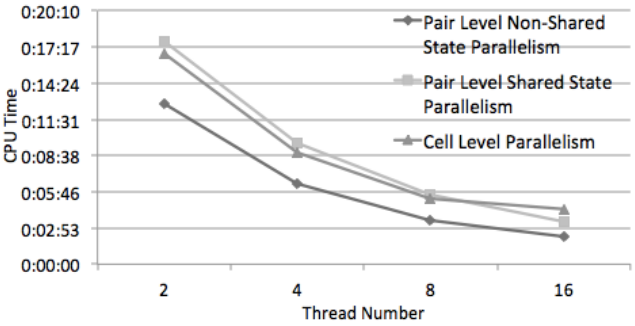
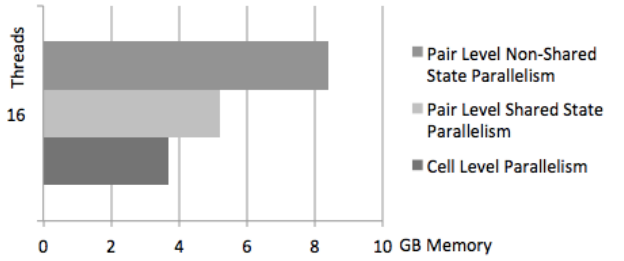


Figure 10: Memory Footprint Comparison



6 Conclusions

Feature extraction procedure in NLP system can achieve significant performance boost by parallelization. In data level parallelism, training time can be reduced. Furthermore, response time can be reduced by implementing parallel parsing algorithm. Parallel parsing algorithm needs careful examination of the dependency among computations. Based on this, lock-free algorithm can be implemented in a bottom-up approach. Beyond that, shared data access can also be optimized as discussed in Section[5.6.2].

But to further improve the performance, the software and hardware configuration need to be take into consideration. Trade-offs of memory and CPU should be made to maximize the performance. One example is as discussed in Section[5.6.2], avoiding shared data access leads to higher memory usage which may downgrade the performance of the system when memory is limited. With the development of multicore hardware, parallelizing existing NLP application can greatly improve the performance. It requires reorganization of the computations and knowledge about the software/hardware configuration to better design the program.