CS540 Parallelizing TextDiff Project Report

Silviu Fodor, Xinjiang Shao, Neha Shah, Anshika Agarwal
Oct. 18, 2012

Introduction

TextDiff is a program to compare two text files and report the differences. Our code source is downloaded from [1]. As the author of the project said, the algorithm (but no code) was taken from Java code by Ian F. Darwin, ian@darwinsys.com, January, 1997. Darwin's code was a translation of a C program by D. C. Lindsay, C (1982-1987). [1]

The algorithm is:

1. For each unique line of text create a symbol. The symbol state is: OldOnly, NewOnly, UniqueMatch (both files exactly once), or Other.
2. For each line, create a LineInfo object. Set state = symbol state and establish bidirectional links between UniqueMatch lines in the two files.
3. For each UniqueMatch in old create a "match block". Stretch match blocks forward and backward to include matching lines with any state, including other match blocks.
4. Build a Report of edit commands that can be used to transform Old into New. Matching blocks generate match or move commands. Non-matching blocks generate insert, append, delete or change commands.
5. Iterate the commands to generate a report.

And our goal is to convert the original project into multi-threaded project without changing the result of the program. It means we have to generate the same report with our project regardless of the bug of the original program. Hopefully, it will have better performance with multi-thread. In fact, we have achieved 9.7% performance increase with two large files comparison.

Our project is hosted in github.com. So if you need to check out the source code, you could visit https://github.com/soleo/TextDiff to get the test cases, code and documentation.

Tools used to parallelize the code in Java

- Threadpool - Java provides a library called **java.util.concurrent** in which there is a interface ExecutorService. Thus, we could use it to create a thread pool with the amount of threads. In this way, we could reduce the overhead of the threads creating process, and manage the threads easily.
- Synchronized statement - guarantees mutual exclusion between concurrent threads
- Runnable and Callable interfaces - the instances of the classes that implement these interfaces can be run within different threads.

Parallelizing opportunities

By analysing the flow of the program, we found:

Original Sequential Execution	Modified Parallel Execution
1. read File1 and return String[] 1 2. read File1 and return String[] 2 3. create FileInfo object for String[] 1	1. read both files using two Callables, each run by a thread, that return two FileInfo objects

4. create FileInfo object for String[] 1	*Note: sequential steps 1,2 and 3,4 run in parallel, parallelization is total*
5. populate HashMap<line,Symbol> for File1 6. populate HashMap<line,Symbol> for File2	2. populate HashMap<line,Symbol> using two threads *Note: the synchronized statement on the HashMap bottlenecks the parallel execution, computation done in parallel is minimal*
7. create LineInfo for File1 8. create LineInfo for File2	3. create LineInfo in parallel using two threads *Note: the synchronized statement on the HashMap bottlenecks the parallel execution, computation done in parallel is minimal*
9. for x = 1 to N a. compute stretch match for line x, File1, File2, Forward b. compute stretch match for line x, File1, File2, Forward	4. for x = 1 to N a. compute stretch match for line x, File1, File2, Forward in Thread1 b. compute stretch match for line x, File1, File2, Forward in Thread2 *Note: steps a and b execute in parallel*
10. return the Report (file or console)	5. return the Report (file or console) *Note: executed in one thread*

Trouble shooting and findings

- File Lost (Incomplete project)

  The source code downloaded from [1] is not complete, then we find the missing file TextFileIn.java from Google Code(g-beehive project)[3].
- Bug

  The original project has one bug we found when we are doing group discussion. TextDiff cannot handle cases like following:

Old File	New File

One Two Two	Two Two

The result from running the program is:

```
Change Old line(s) 1-3
  1: one
  2: two
  3: two
New line(s) 1-2
  1: two
  2: two
```

In fact, the correct result should not be like this. The second line and third line from old file is the same with the first line and second line in new file. So the correct answer for old file and new file should indicate “one” is removed. If remove duplicate “two” in both files, the result is correct and normal.

This bug is introduced by using SymbolCollection Class which is a HashMap. The key for SymbolCollection is the line string, thus if we have same lines old file and new file, they will not act correctly because of the same key.

- Compiler Optimization

In the first run, we saw a tremendous improvement, almost 50%, of our multithreaded version program, but the good result happened because we didn’t turn off Java Compiler Optimization. So the first time we can a method in class, it is slow, but the second time will be super fast since java virtual machine will use the binary code from the first time we used. And in our project, we have some java class used in both sequential version and parallel version. By adding flag “-Djava.compiler=NONE”, we could disable the Java Compiler Optimization to get better measurement.[5]

- System.nanoTime() Vs. System.currentTimeMillis()

There are a lot of discussion about choosing the right function to do time measurement in program[6]. nanoTime() provides nanosecond precision, but not necessarily nanosecond accuracy[7]. This statement is confusing for us. In other words, nanoTime() is platform-specific, so it behaves differently on different OS/Hardware. And it is reported that the method failed to get the right time duration when you’re using Windows with multi-core hardware. At the same time, currentTimeMillis() has less precision. So the best way here is to run the program in different platform to see if there is big difference by using nanoTime().

Testing and evaluation

The test machine we used is MacBook Pro 7.1. The configuration of the machine is as follows.

- **Processor** 2.66 GHz Intel Core 2 Duo
- **Memory** 4 GB 1067 MHz DDR3
- **Software** OS X 10.8.2 (12C60)
- **Java Version** 1.6.0_35

Table 1. Performance Result from two small files(20kb)

No.	Sequential-Run Time (ms)	Parallel-Run Time (ms)	Improvement (%)
1	74.080	81.948	-10.62
2	65.743	71.411	-8.62
3	72.689	71.723	1.33
4	66.107	70.646	-6.86
5	71.564	67.887	5.13

6	71.865	67.511	6.05
7	68.878	64.704	6.05
8	70.574	73.305	-3.86
9	76.187	72.816	4.42
10	67.165	73.992	-10.16

Average improvement from parallelism is -1.71%.

Table 2. Performance Result from two medium files(274kb)

No.	Sequential-Run Time (ms)	Parallel-Run Time (ms)	Improvement (%)
1	1247.039	1214.575	2.60
2	1184.331	1198.949	-1.23

3	1183.347	1157.861	2.15
4	1202.100	1166.772	2.93
5	1230.590	1159.893	5.74
6	1209.030	1197.008	0.99
7	1197.240	1181.876	1.28
8	1191.395	1179.696	0.98
9	1184.276	1178.951	0.44
10	1193.732	1166.387	2.29

Average improvement from parallelism is 1.82%.

Table 3. Performance Result from two large files(881kb)

No.	Sequential-Run Time (ms)	Parallel-Run Time (ms)	Improvement (%)
1	1757.516	1587.132	9.69
2	1760.519	1599.211	9.16
3	1768.623	1605.765	9.20
4	1721.552	1630.607	5.28
5	1724.718	1604.238	6.98
6	1735.584	1597.761	7.94
7	1758.999	1627.380	7.48
8	1742.433	1583.542	9.11
9	1741.531	1632.383	6.26
10	1739.385	1625.055	6.57

Average improvement from parallelism is 7.77%.

Conclusion

The algorithm of TextDiff uses one data structure for all calculations, which imposes a bottleneck for parallelization. Reading of the files and blocks comparison where the only executions that we managed to implement 100% in parallel. The performance of our parallel implementation increases proportional with the size of the files we compare. As we can see above, we obtained a maximum increase of about 8% for files of size 0.8MB.

Reference

1. http://www.surfscranton.com/architecture/TextDiff.htm
2. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html
3. http://code.google.com/p/g-beehive/source/browse/trunk/www/app/textdiff/TextFileIn.java
4. C.T.H. Everaars, F. Arbab and B. Koren. "Using coordination to restructure sequential source code into a concurrent program," *Proceedings IEEE International Conference on Software Maintenance*, pp. 342—351, 2001.
5.http://stackoverflow.com/questions/11229291/java-method-in-static-initialization-block-is-slower-than-in-main-methd
6. http://stas-blogspot.blogspot.com/2012/02/what-is-behind-systemnanotime.html
7. http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html#nanoTime()