Report for the Java/C++ Concurrency Benchmark Topic Paper #22

Alex Laird
CS-3210-01
4/27/09
Survey of Programming Languages
Spring 2009
Computer Science, (319) 360-8771
alexdlaird@cedarville.edu

Grading Rubric		Max	Earn
	On Time/Format	1	
	Correct	5	
	Clear	2	
	Concise	2	
	Total	10 pts	

Peer Reviewer

ABSTRACT

This paper discusses the thread-based matrix multiplier implemented in Java and C++ for benchmarking purposes.

Keywords
Java, C++, Matrix, Thread, Concurrency, Benchmark

1. INTRODUCTION

Concurrency, one of the most overlooked concepts in low-level programming today, is the ability to complete multiple processes simultaneously. Both Java and C++ have support for concurrency on some level through the use of threads, though attaining this level of optimal performance may be more difficult to implement in one language over the other.

To prove the existence of concurrency in both languages, we constructed a simple *n* by *n* square matrix multiplier that would generate *n* threads to multiple each row and column of the two matrices together. The project specifications detailed that the benchmark should be written for 6 by 6 matrices, but it easier and more versatile to write it for an *n* by *n* matrix and simply define a global variable *n* to be 6.

The user has the ability to specify their own matrix or simply type "default" to run the benchmark with the default matrix. The defulat matrix is shown below:

```
[3 4 6 2 5 2]
[1 8 3 3 7 0]
[0 10 7 9 3 5 ]
[5 0 1 4 3 0]
[0 0 0 0 7 8]
[6 6 4 10 4 3]
```

2. JAVA IMPLEMENTATION

Java, unlike C++, has threads built directly into the standard library two separate ways. Implementation of threads can be achieved through the Thread object or through the Runnable object. We chose the Thread object and abstracted the actual matrix multiplication part out to a separate class that extended Thread. For each of the *n* row by column multiplications, a separate thread was created to do the multiplication. Java has an extremely simple and very fast implementation. The actual thread instantiation is seen in multiply() on lines 85-87. The thread itself is in Multiplication.java.

3. C++ IMPLEMENTATION

The C++ benchmark uses pthreads, which are default includes in a Unix-based system but may require additional libraries upon compile if run in Windows. Just like in the Java benchmark, we abstracted the actual multiplication process out to call it repeatedly, once for each of the *n* row by column multiplications. The actually thread instantiation is seen in multiply() on line 112. The thread itself is mThread() on lines 67-78.

4. OUTPUT

The default matrix was used both in the C++ and in the Java benchmarks, and the matrix was simply multiplied by itself. The output received was:

```
[45 112 126 56 145 36]
[15 224 63 84 203 0]
0 280 147 252 87 90]
[75 0 21 112 87 0]
[0 0 0 0 203 144]
[0 0 0 0 0 0]
```

In Java, the multiplication method had a runtime of 16 milliseconds, and the total runtime of the entire program was also 16 milliseconds. In C++, the multiplication method had a runtime of 609 milliseconds, and the total runtime of the entire program was 631 milliseconds.

5. CONCLUSIONS

Due to the additional hassle of configuring pthreads for such a simple task as matrix multiplication, and since threads are a default library in any version of the JDK, the benchmark showed that Java was significant faster than C++. On a larger scale, the efficiency of C++ over Java may start to show when using threads, but on such a small project, Java was far more efficient in thread usage, both in programming and in performance.