Analysis of Vector Addition with Threads

Arthur Ceccotti - 8544173

02-03-2016

I have written an implementation of addition of two vectors of random contents using Java (source code on page 3). In order to assure correctness of the program, I approached development using the Test Driven Agile methodology with JUnit tests (attached with the code).

Upon completing the application, I continued by evaluating its performance, running it multiple times with different parameters (number of threads and vector size). By attempting to soften intermittent performance drops/spikes, the program was run 20 times, allowing the gathering of average values and standard deviation.

At first I thought the increase of threads, given constant vector size, would result on a linear increase of performance. Because each thread has equal load (with at most one more operation), vector addition with 4 threads would ideally run twice faster than that with 2 threads of same size. Thus my expected speedup was of n times compared to a fully sequential execution, where n is the number of active threads.

This would certainly not be the case for small vector inputs, given that thread initialisation would be a large overhead. Another issue would be executing the program on a single core machine, as threads would never run *truly in parallel* and context switching would be an additinal cost. Figure 1 describes the performance decrease given increase of number of threads for a small problem size of 10,000 elements on the machine *mcore48.cs.man.ac.uk* [1]. I evaluate this to occur mostly due to the thread initialisation cost. As we can see, the program performs much better with a single thread for this problem size, which means in this case it is quicker to do the whole vector addition sequentially, rather than spending a long time spawning threads which will die after doing few operations.

[1] Dell/AMD quad 12-core (48 cores total), each with specs:
model name: AMD Opteron(tm) Processor 6174
clock speed: 2.2 GHZ
cache size: 512 KB

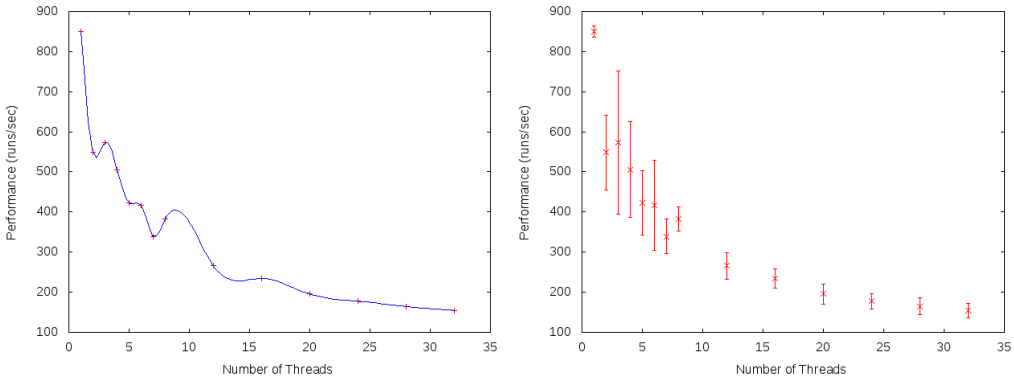

Figure 1: Parallel vector addition of size 10,000. Average performance (left) and standard deviation error range (right) from 20 runs

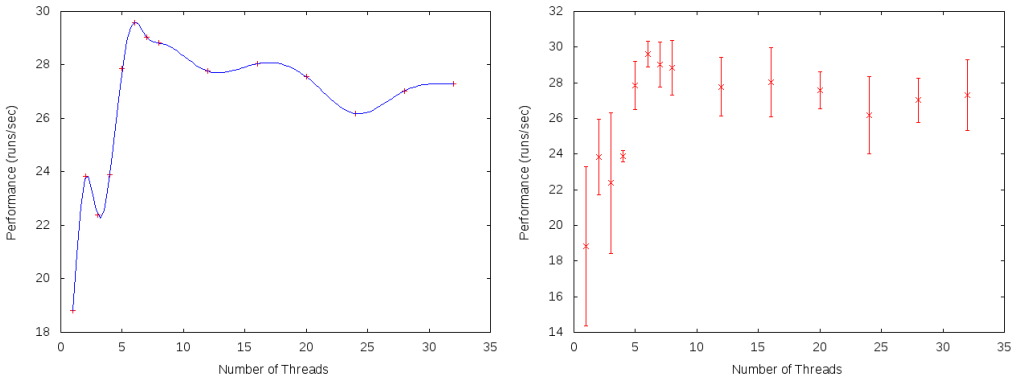

Figure 2: Parallel vector addition of size 10,000,000. Average performance (left) and standard deviation error range (right) from 20 runs

For larger problem sizes (>1,000,000), thread initialization of a few milliseconds starts becoming insignificant compared to the lifetime of each thread. The plot in figure 2 describes the run of my program for the vector size of 10,000,000 with variable threads, where such performance increase is clearly visible.

This now shows how advantageous multithreaded programming can be for large input problems when few data dependencies are present.

```java
import java.util.Random;

public class VectorAdder {

    public static final Random random = new Random();

    public static void main(String[] args){
        if(args.length != 2){
            System.err.println("Exactly two arguments are needed.");
            return;
        }

        final int NUMBER_OF_THREADS = Integer.parseInt(args[0]);
        final int VECTOR_LENGTH = Integer.parseInt(args[1]);

        int[] A = randomVector(VECTOR_LENGTH);
        int[] B = randomVector(VECTOR_LENGTH);

        add(A,B, NUMBER_OF_THREADS);
    }

    public static int[] add(int[] A, int[] B, int NUMBER_OF_THREADS){
        if(A == null || B == null){
            return null;
        }

        if(A.length != B.length){
            throw new IllegalArgumentException(
              "Sizes of both arrays must be the same.");
        }

        if(NUMBER_OF_THREADS <= 0){
            throw new IllegalArgumentException(
              "Must have at least 1 thread.");
        }

        int VECTOR_LENGTH = A.length;
        int[] result = new int[VECTOR_LENGTH];

        Thread[] threads = new Thread[NUMBER_OF_THREADS];

        int threadLoad = VECTOR_LENGTH / NUMBER_OF_THREADS;
        int leftOvers = VECTOR_LENGTH - (threadLoad * NUMBER_OF_THREADS);

        //Spread load equally to the different threads such that they
        //process the same amount of elements (or one more)
        int i = 0;
        int j = threadLoad;
        for(int t = 0; t < NUMBER_OF_THREADS; t++){
```

```
            if(leftOvers > 0){
                leftOvers--;
                j++;
            }

            threads[t] = new VectorAdderThread(i, j, A,B, result);
            i = j;
            j += threadLoad;
        }

        long startTime = System.nanoTime();
        for(Thread t : threads){
            t.start();
        }

        for(Thread t : threads) {
            try {
                t.join();
            }
            catch (InterruptedException e){
                System.err.println("Thread " + t + " was interrupted!");
            }
        }

        long endTime = System.nanoTime();

        //Calculate performance as 1 / execution_time_in_seconds
        System.out.println(1000000000.0/(endTime - startTime));

        return result;
    }

    public static int[] randomVector(int l){
        if(l < 0){
            return null;
        }

        int[] V = new int[l];

        for(int i=0; i < l; i++){
            V[i] = random.nextInt();
        }

        return V;
    }


}
```

```
public class VectorAdderThread extends Thread {

    private int i, j; //start and end index in the array
    private final int[] A, B, C; //input arrays A,B and output array C

    public VectorAdderThread(int i, int j, int[] A, int[] B, int[] C) {
        this.i = i;
        this.j = j;
        this.A = A;
        this.B = B;
        this.C = C;
    }

    @Override
    public void run() {
        //Perform vector addition for i-j section
        for(;i<j;i++){
            C[i] = A[i] + B[i];
        }
    }
}
```

```
import org.junit.Test;

import static org.junit.Assert.*;

public class VectorAdderTest {

    @Test
    public void testAddNull() {
        assertNull(VectorAdder.add(null, null, 1));
    }

    @Test(expected=IllegalArgumentException.class)
    public void testAddDifferentSizes() {
        VectorAdder.add(new int[1], new int[2], 1);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testAddZeroThreads() {
        VectorAdder.add(new int[5], new int[5], 0);
    }

    @Test
    public void testAddEmpty() {
        assertEquals(
          VectorAdder.add(new int[0], new int[0], 1).length, 0);
    }
```

```
@Test
public void testAddOneEntryOneThread() {
    assertArrayEquals(
      VectorAdder.add(new int[]{1}, new int[]{2}, 1), new int[]{3});
}

@Test
public void testAddOneEntryTwoThreads() {
    assertArrayEquals(
      VectorAdder.add(new int[]{1}, new int[]{2}, 2),
                    new int[]{3});
}

@Test
public void testAddTwoEntriesTwoThreads() {
    assertArrayEquals(
      VectorAdder.add(new int[]{1,2},new int[]{2,-2}, 2),
                    new int[]{3,0});
}

@Test
public void testAddFiveEntriesThreeThreads() {
    assertArrayEquals(
      VectorAdder.add(
        new int[]{1,2,3,4,5}, new int[]{2,-2,1,2,3}, 3),
        new int[]{3,0,4,6,8});
}

@Test
public void testAddThousandEntriesOneThread() {
    int[] A = dummyArray(1000);
    int[] B = dummyArray(1000);

    assertArrayEquals(VectorAdder.add(A,B, 1), expectedAdd(A,B));
}

@Test
public void testAddThousandEntriesTwoThreads() {
    int[] A = dummyArray(1000);
    int[] B = dummyArray(1000);

    assertArrayEquals(VectorAdder.add(A,B, 2), expectedAdd(A,B));
}

@Test
public void testAddThousandEntriesFivehundredThreads() {
    int[] A = dummyArray(1000);
    int[] B = dummyArray(1000);
```

```
        assertArrayEquals(VectorAdder.add(A,B, 500), expectedAdd(A,B));
    }

    @Test
    public void testAddThousandEntriesThousandAndOneThreads() {
        int[] A = dummyArray(1000);
        int[] B = dummyArray(1000);

        assertArrayEquals(VectorAdder.add(A,B, 1001), expectedAdd(A,B));
    }

    private int[] dummyArray(int l){
        int[] D = new int[l];

        for(int i = 0; i < l; i++) {
            D[i] = i;
        }

        return D;
    }

    private int[] expectedAdd(int[] A, int[] B){
        int[] result = new int[A.length];

        for(int i = 0; i < result.length; i++){
            result[i] = A[i] + B[i];
        }

        return result;
    }
}
```