Analysis of Merge Sort with Threads

Arthur Ceccotti - 8544173

14-04-2016

1 Introduction

I have written a version of Merge Sort of a vector of random integer contents in Java. The correctness of the program was guaranteed by running it against a sequence of JUnit tests developed in advance (attached on the bottom of this report under listing 1).

Upon completion of the application, I evaluated its performance by plotting it with different parameters (vector size and number of threads). Intermittent performance drops/spikes were reduced by running every instance 10 times, allowing the gathering of average values and standard deviation. The large amount of data produced from all the runs was analysed and plotted by my Python script under listing 5. The data plots were a consequence of running the given program on *mcore48.cs.man.ac.uk*. [1]

2 Evaluation

On an instance of this problem, one would expect linear speed up of performance with the increase of threads, as load is equal for every thread and there are enough hardware processors. This means, with the same vector size, 10 threads should run twice faster than 5 threads. This O(n) speedup is certainly not true for the sorting of small vectors, because of overheads such as **thread initialisation time** (which includes OS calls, stack memory assignment for the thread, adding it to the JVM data structure,...) and **joins**. This meas the overall thread creation bottleneck may take longer than the sorting itself. This is a very similar behaviour to the previous coursework for Vector Addition with Threads.

For small vector sizes, the increase of number of threads to solve the problem in fact decreases performance, as can be seen on figures 1 and 2 for vector sizes of *1000* and *10,000*. In these cases the program has its best performance with a

[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: 1,000 elements

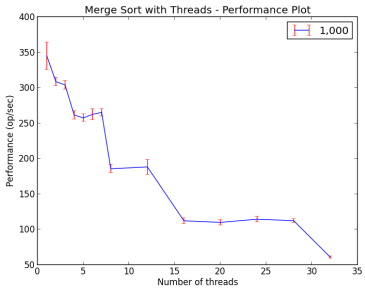


Figure 2: 10,000 elements

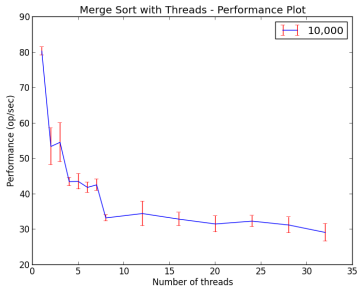


Figure 3: 100,000 elements

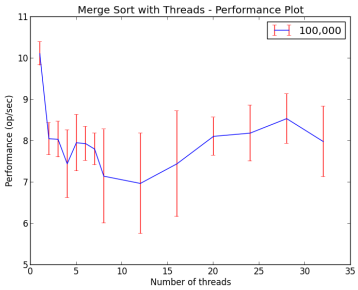
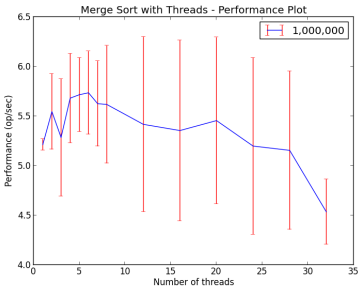


Figure 4: 1,000,000 elements



single thread sorting the whole array sequentially, which saves having to spawn threads which will take milliseconds to initialise and quickly die after doing very few operations.

The balance of thread overhead and speedup is reached at *100,000-1,000,000* vector sizes, as seen on figures 3 and 4 respectively, where adding more threads does not significantly increase performance, as it introduces similar slowdown. Because of this the graphs present wide error bars and an overall horizontal performance.

The advantage of threads can be finally seen on large instances of the problem with *>10,000,000* elements, as described on figures 5 and 6. Joins and initialisation still pose as an overheads, but are insignificant given the amount of sorting performed.

Figure 5: 10,000,000 elements

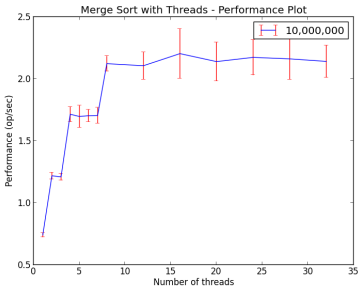
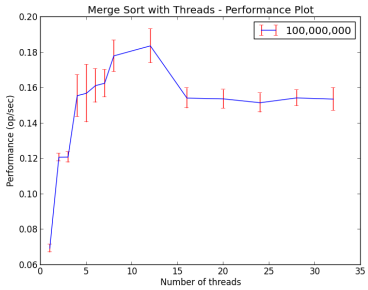


Figure 6: 100,000,000 elements



3 Code

3.1 Java Unit Tests

Listing 1: MergeSort Unit Tests

```
import org.junit.Test;

import static org.junit.Assert.*;

public class MergeSortTest {

    @Test
    public void testNullArrayOneThread(){
        assertNull(MergeSortThread.sort(null,1));
    }

    @Test(expected=IllegalArgumentException.class)
    public void testSortArrayZeroThreads(){
        MergeSortThread.sort(new int[]{1,2,3},0);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testSortArrayNegativeThreads(){
        MergeSortThread.sort(new int[]{1,2,3},-1);
    }

    @Test
    public void testSortEmptyArrayOneThread(){
        assertEquals(MergeSortThread.sort(new int[0],1).length, 0);
    }

    @Test
    public void testSortSingleElementArrayOneThread(){
```

```
    assertEquals(MergeSortThread.sort(new int[]{123},1)[0], 123);
}

@Test
public void testSortSingleElementArrayTwoThreads(){
    assertEquals(MergeSortThread.sort(new int[]{123},2)[0], 123);
}

@Test
public void testSortTwoElementsArrayOneThread(){
    int[] A = new int[]{1,-1};
    assertSorted(MergeSortThread.sort(A,1));
}

@Test
public void testSortTwoElementsArrayTwoThreads(){
    int[] A = new int[]{1,-1};
    assertSorted(MergeSortThread.sort(A,2));
}

@Test
public void testSortTwoElementsArrayThreeThreads(){
    int[] A = new int[]{1,-1};
    assertSorted(MergeSortThread.sort(A,2));
}

@Test
public void testSortShortArrayInvertedTwoThreads(){
    int[] A = new int[]{100,50,20,10,5,3,1,-1};
    assertSorted(MergeSortThread.sort(A,2));
}

@Test
public void testSortShortArraySortedTwoThreads(){
    int[] A = new int[]{1,2,3,4,5};
    assertSorted(MergeSortThread.sort(A,2));
}

@Test
public void testSortRandomArrayOneThread(){
    int[] A = MergeSort.randomArray(100);
    assertSorted(MergeSortThread.sort(A,1));
}

@Test
public void testSortRandomArrayTwoThreads(){
    int[] A = MergeSort.randomArray(100);
    assertSorted(MergeSortThread.sort(A,2));
}
```

```
@Test
public void testSortRandomArrayFiveThreads(){
    int[] A = MergeSort.randomArray(100);
    assertSorted(MergeSortThread.sort(A,5));
}

@Test
public void testSortRandomArrayFiftyThreads(){
    int[] A = MergeSort.randomArray(100);
    assertSorted(MergeSortThread.sort(A,50));
}

@Test
public void testSortRandomArraySeventyFiveThreads(){
    int[] A = MergeSort.randomArray(100);
    assertSorted(MergeSortThread.sort(A,75));
}

@Test
public void testSortRandomArrayHundredThreads(){
    int[] A = MergeSort.randomArray(100);
    assertSorted(MergeSortThread.sort(A,100));
}

@Test
public void testSortRandomArrayHundredOneThreads(){
    int[] A = MergeSort.randomArray(100);
    assertSorted(MergeSortThread.sort(A,101));
}

@Test
public void testSortLongRandomArrayOneThread(){
    int[] A = MergeSort.randomArray(10000);
    assertSorted(MergeSortThread.sort(A,1));
}

@Test
public void testSortLongRandomArrayThreeThreads(){
    int[] A = MergeSort.randomArray(10000);
    assertSorted(MergeSortThread.sort(A,3));
}

@Test
public void testSortLongRandomArrayEightThreads(){
    int[] A = MergeSort.randomArray(10000);
    assertSorted(MergeSortThread.sort(A,8));
}

@Test
public void testSortTenMillionEntriesRandomArrayEightThreads(){
```

```
        int[] A = MergeSort.randomArray(10000000);
        assertSorted(MergeSortThread.sort(A,8));
    }

    private static void assertSorted(int[] A){
        assertTrue(isSorted(A));
    }

    private static boolean isSorted(int[] A){
        for(int i=0; i < A.length-1; i++){
            if (A[i] > A[i+1]) {
                return false;
            }
        }
        return true;
    }
}
```

3.2 Java Merge Sort Program

Listing 2: MergeSort Main class

```
import java.util.Random;

public class MergeSort {

    private static final Random random = new Random();

    public static void main(String[] args){
        if(args.length != 2){
            System.err.println("There must be exactly two arguments.");
            System.exit(1);
        }

        /* arguments [0] - length of random int array to sort
                     [1] - number of threads to execute sorting */

        int arrayLength = 0;
        try{
            arrayLength = Integer.parseInt(args[0]);
        }
        catch(NumberFormatException e){
            System.err.println("Array length must be an integer.");
            System.exit(1);
        }

        if(arrayLength < 0){
            System.err.println("Array length must be positive or zero");
```

```
            System.exit(1);
        }

        int numberOfThreads = 0;
        try{
            numberOfThreads = Integer.parseInt(args[1]);
        }
        catch(NumberFormatException e){
            System.err.println("Number of threads must be an integer.");
            System.exit(1);
        }

        if(numberOfThreads <= 0){
            System.err.println("Number of threads must be positive.");
            System.exit(1);
        }

        //generate random dummy array to produce sorting benchmark
        int[] A = randomArray(arrayLength);

        long startTime = System.nanoTime();
        MergeSortThread.sort(A, numberOfThreads);
        long endTime = System.nanoTime();

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

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

        int[] A = new int[l];

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

        return A;
    }
}
```

Listing 3: MergeSort Thread class

```
import java.util.Arrays;

public class MergeSortThread extends Thread{
```

```
/*
    i = starting index for this thread to sort, inclusive
    j = ending index for this thread to sort, non-inclusive
    numberOfThreads = total number of children (directly or
        indirectly)
                    spawned by this instance
    A = int array to sort in-place
*/

private int i, j, numberOfThreads;
private int[] A;

private MergeSortThread(int[] A, int i, int j, int numberOfThreads) {
    this.A = A;
    this.i = i;
    this.j = j;
    this.numberOfThreads = numberOfThreads;
}

public static int[] sort(int[] A, int numberOfThreads){
    //safety checks to pass the unit tests :)
    //won't add significant time to the benchmark
    if(A == null){
        return null;
    }

    if(numberOfThreads <= 0){
        throw new
               IllegalArgumentException("At least one thread" +
                                        " is required to sort.");
    }

    return sort(A, 0, A.length, numberOfThreads);
}

//statically invokes given number of threads to sort array
private static int[] sort(int[] A, int i, int j, int
     numberOfThreads){
    if(numberOfThreads > 1){
        // spawn a thread to sort the first half
        Thread t = new MergeSortThread(A, i, (i+j)/2,
            numberOfThreads/2);
        t.start();

        //recursively sort the second half
        sort(A, (i+j)/2, j, numberOfThreads/2);

        try {
            t.join();
        }
```

```
        catch (InterruptedException e){
            System.err.println("Oh well...");
        }

        /* After both threads have sorted their sections, merge
           them (knowing they are separated at (i+j)/2).
           The merge cannot be done in-place, so copy it back to A */
        int[] m = merge(A, i, j);

        System.arraycopy(m,0,A,i,m.length);
    }
    else{
        /* Base case: there are no threads left to spawn, thus the
           current thread must sort its section of the array
               sequentially */
        Arrays.sort(A, i, j);
    }

    return A;
}

@Override
public void run() {
    sort(A, i, j, numberOfThreads);
}

private static int[] merge(int[] A, int l, int r) {
    int[] M = new int[r-l];

    int leftLim = (l+r)/2; //separation between subarrays

    int i1 = l; //current index on first half
    int i2 = leftLim; //current index on second half
    int m = 0; //current index on merged array

    while (i1 < leftLim && i2 < r){
        if (A[i1] < A[i2]) {
            M[m] = A[i1];
            i1++;
        }
        else {
            M[m] = A[i2];
            i2++;
        }
        m++;
    }

    /* if there are any elements left on the first
       or second half, move them to the merged array */
    System.arraycopy(A, i1, M, m, leftLim - i1);
```

```
        System.arraycopy(A, i2, M, m, r - i2);

        return M;
    }
}
```

3.3 Data Evaluation Scripts

Listing 4: Bash scripts to run MergeSort with multiple parameters

```
#!/bin/sh
#
# request Bourne shell as shell for job
#$ -S /bin/sh
#
# use current working directory
#$ -cwd
#
# join the output and error output in one file
#$ -j y
#
# use the par environment for parallel jobs
#$ -pe par 32
#
# set up affinity mask (defines cores on which to run threads)
# default mask

#echo "length,threads,performance"
for l in 1000 10000 100000 1000000 10000000 100000000
do
  for p in 1 2 3 4 5 6 7 8 12 16 20 24 28 32
  do
   for times in {1..10}
   do
      performance=$(numactl
          --physcpubind=0,1,2,4,5,6,8,9,10,12,13,14,16,17,18,20,21,22,24,25,26,28,29,31,32,33,34,36,37
          java MergeSort $l $p)
      echo "$l,$p,$performance"
    done
    echo ""
  done
done
```

Listing 5: Python scripts to gather and plot results

```
import sys
import matplotlib.pyplot as plt
#import statistics
```

```
def mean(data):
    """Return the sample arithmetic mean of data."""
    n = len(data)
    if n < 1:
        raise ValueError('mean requires at least one data point')
    return sum(data)/n # in Python 2 use sum(data)/float(n)

def _ss(data):
    """Return sum of square deviations of sequence data."""
    c = mean(data)
    ss = sum((x-c)**2 for x in data)
    return ss

def stdev(data):
    """Calculates the population standard deviation."""
    n = len(data)
    if n < 2:
        raise ValueError('variance requires at least two data points')
    ss = _ss(data)
    pvar = ss/n # the population variance
    return pvar**0.5


def plotandsave(l,stats):
    plt.errorbar(stats["threads"], stats["averages"],
        yerr=stats["stdevs"], ecolor='r', label="{:,d}".format(l))
    plt.legend(loc='upper right')
    plt.xlabel('Number of threads')
    plt.ylabel('Performance (op/sec)')
    plt.title('Merge Sort with Threads - Performance Plot')
    plt.savefig('graphs/{}.png'.format(l))
    plt.show()

statsArray = []

print("length,threads,avg,stdev")

#stats array has format: length,threads,performance

import csv
with open(sys.argv[1], 'rb') as f:
    reader = csv.reader(f)

    l = 0
    statsArray = []

    for row in reader:
      if not row: #reached separation
            avg = mean(list(float(r[2]) for r in statsArray))
```

```
        std = stdev(list(float(r[2]) for r in statsArray))

        stats["threads"].append(int(statsArray[0][1]))
        stats["averages"].append(avg)
        stats["stdevs"].append(std)

        print("{},{},{},{}".format(statsArray[0][0],
             statsArray[0][1], avg, std))

        del statsArray[:]
    else:
        if int(row[0]) != l:
            if l is not 0:
                   plotandsave(l,stats)
            l = int(row[0])
            stats = {"threads" : [], "averages" : [], "stdevs" : []}

        statsArray.append(row)

plotandsave(l,stats)
```