Analysis of Poisson Relaxation with Threads

Arthur Ceccotti - 8544173

May 6, 2016

1 Introduction

I have written a version of the *Poisson's Equation* using the relaxation technique with a variable number of threads. Correctness of the program was guaranteed by checking the expected number of iterations (65,251) and also manually visualizing the array convergence over different iterations, as seen in figure 1, produced from listings 7 and 6.

Intermittent performance drops/spikes were reduced by running every instance 10 times on *mcore48.cs.man.ac.uk*[1], allowing the gathering of average values and standard deviation. The data produced from all the runs was analysed and plotted by a Python script under listing 5.

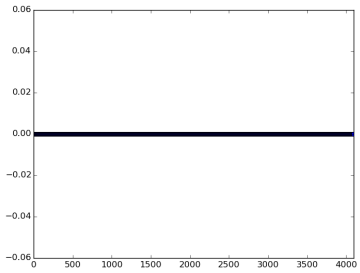
2 Evaluation

Figure 2 plots the average performance and error bars (based on standard deviation) of the program for this constant vector size. As can be seen in the figure, the overall performance drops drastically with the addition of new threads, which would be counter intuitive to the advantages of multithreaded programs. I evaluate that the vector size (or accuracy of the problem) is too small (only about 4,000) to allow improvements from multithreading. As mentioned in earlier courseworks, **thread initialization** is expensive (processor-wise and memory-wise) and so is **blocking at the join**, meaning for small instances such bottleneck may take longer than a single thread solving the problem by itself, which is exactly the case here.

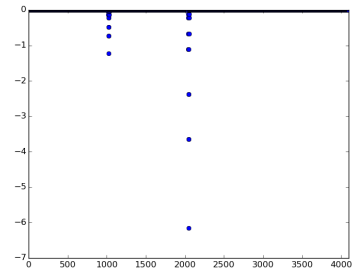
In this program we need to use a **CyclicBarrier** to make sure all threads are ready to proceed (avoiding writing to boundary values while they are being read), which highly impacts performance, similarly to the join. I initially wrote a

[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

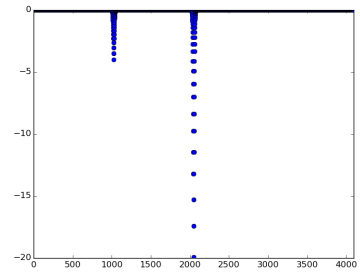
Figure 1: Approximation of Poisson's Equation over different iterations



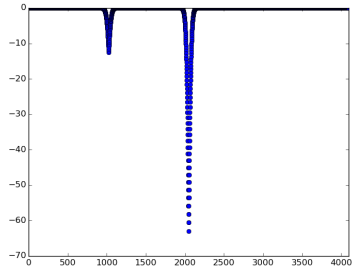
0 iterations



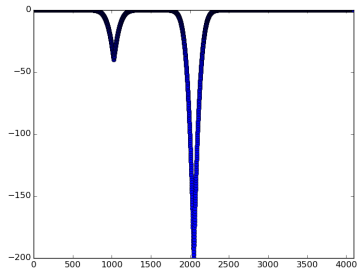
10 iterations



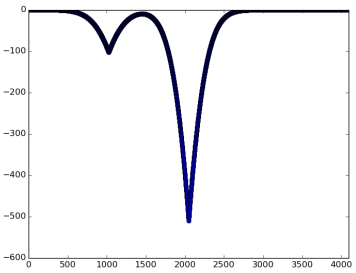
100 iterations



1,000 iterations

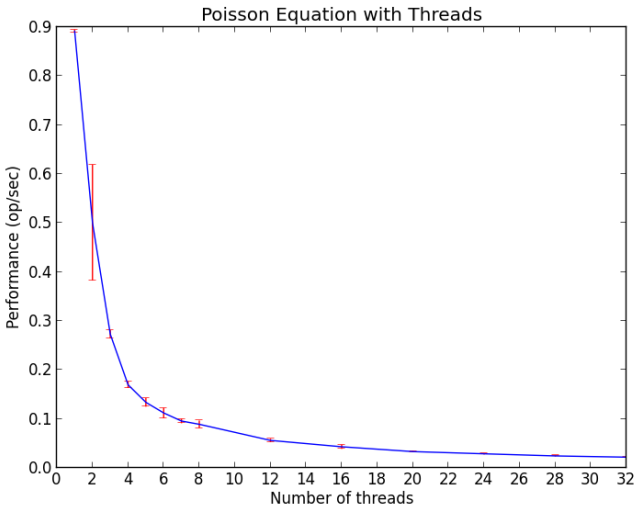


10,000 iterations



65,250 iterations

Figure 2: Performance plot



solution in which each thread awaited twice at the barrier at each iteration[2], but that introduced even larger bottleneck, as threads were doing very little work before reaching a check-point to wait for other threads. This was improved by passing a Runnable to be executed upon tripping of the barrier, allowing the use of a single await at each iteration (see code at listing 3). The comparison of both implementations is shown in figure 3, in which the speedup advantages of using a single await per iteration is clearly visible.

Another enhancement was made by having each thread checking for convergence of their own regions concurrently and if at least one thread had not converged, it would vote to have another iteration, which would be accepted by all threads. This is much faster than having a single thread scanning the whole array, checking for convergence, which completely defeats the purpose of using threads.

Although these enhancement may be barely visible with such a small array (about 4,000 elements), I evaluate that my technique, using *one await per iteration* and *multithreaded convergence check* would perform very well for large vectors.

A factor which could also impact performance is **cache misses for large problems**, specially given we are using arrays of doubles (64-bit elements). If the working set of a thread cannot always be kept in cache (ie. it is full), cache misses would highly slow down the program. This is very unlikely the case for our current problem, but could impact an array with millions of elements.

A linear, **ideal performance** increase with the addition of threads is shown

[2] Once to read the region of the array, writting to the $newV$ + once to check for convergence and write values back to V

Figure 3: Performance comparison

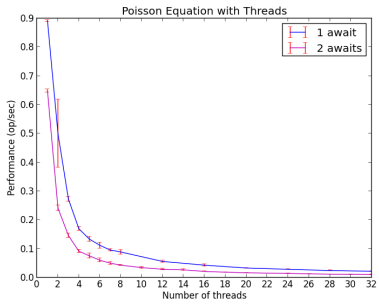
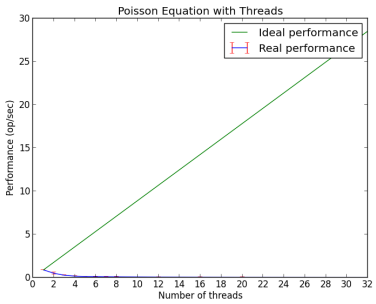


Figure 4: Ideal performance



on figure 4 which assumes that if one thread runs at 1 op/sec, then two threads will run at 2 op/sec. In summary, for the Poisson's Equation problem, as mentioned earlier, this O(n) speedup is surely not realistic due to the following multithreading bottlenecks:

- **Awaiting at barrier**
- **Final join**
- **Convergence voting**
- **Cache misses for large vectors**

3 Code

3.1 Poisson Solver

Listing 1: Poisson Solver

```
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.atomic.AtomicBoolean;

public class PoissonSolver {

    //Initialize force function
    private static final int VECTOR_LENGTH = 4097;
    private static final double[] F = new double[VECTOR_LENGTH];
    static {
        F[1024] = 1.0D;
        F[2048] = 5.0D;
    }

    public static void main(String[] args){

        if(args.length != 1){
            System.err.println("Expected single argument: number of
                threads");
            System.exit(1);
        }

        int nThreads = 0;

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

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

        /*
           V is a wrapper to a double[] (ie. equivalent to C double**),
           representing the approximation to the Poisson's equation
           applied to the Force Function. copyV represents the newV,
           ie. the V at next iteration, which will be swapped with
           V at the end of the iteration.
           Wrappers are used to avoid using System.arraycopy
           and simply use this swapping technique from within
```

```
   the thread, as seen on the barrier runnable on line 59
 */
final ArrayWrapper V = new ArrayWrapper(new
    double[VECTOR_LENGTH]);
final ArrayWrapper copyV = new ArrayWrapper(new
    double[VECTOR_LENGTH]);

/* allConverged is a boolean wrapper, visible to all Poisson
    Threads
   which tells if 'at least one thread has not converged'.
   If allConverged is set, it means we have reached the
   final convergence criteria and then stopSignal is set
   to tell all threads to stop. */
final AtomicBoolean allConverged = new AtomicBoolean(true);
final AtomicBoolean stopSignal = new AtomicBoolean(false);

final Thread[] threads = new Thread[nThreads];

final CyclicBarrier barrier = new CyclicBarrier(nThreads,
       new Runnable() {
           @Override
           public void run() {
               //code run by the last thread to reach barrier

               /*
                  if all threads have converged, ask them
                  to not do any more iterations and stop.
                  Much better performance than having
                  this one thread reading the whole array
                  to check for convergence.
                  Performance! :)
                  */
               if(allConverged.get()){
                   stopSignal.set(true);
                   return;
               }
               else{
                   /* else, reset allConverged for the next
                      iteration, which will later be set to
                      false if at least one thread
                      has not converged */
                   allConverged.set(true);
               }

               /*
                  Swap: the new approximation function is now
                  placed on V (instead of arraycopy at every
                  iteration). Much better performance! :D
                */
               double[] tmp = V.get();
```

```
                V.set(copyV.get());
                copyV.set(tmp);
            }
        });

    final int threadLoad = (VECTOR_LENGTH-2) / nThreads;
    int leftOvers = (VECTOR_LENGTH-2) - (threadLoad * nThreads);

    /* Spread load equally to the different threads such that they
       process the same amount of elements (or one more) */
    int i = 1;
    int j = threadLoad;
    for(int t = 0; t < nThreads; t++){
        if(leftOvers > 0){
            leftOvers--;
            j++;
        }

        /* so many parameters! only other way would be having static
           variables, but that's dirty! */
        threads[t] = new PoissonThread(V, copyV, F, i, j,
                                       barrier, allConverged,
                                           stopSignal);

        i = j+1;
        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();

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

Listing 2: Array Wrapper

```
public class ArrayWrapper{
    private double[] arr;

    public ArrayWrapper(double[] arr){
        this.arr = arr;
    }

    public double[] get() {
        return arr;
    }

    public void set(double[] arr) {
        this.arr = arr;
    }
}
```

Listing 3: Poisson Thread

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.atomic.AtomicBoolean;

public class PoissonThread extends Thread {

    private final double[] F;
    private final ArrayWrapper wrapperV, wrapperCopyV;
    private final int start, end; //start to end inclusive
    private final CyclicBarrier barrier;
    private final AtomicBoolean allConverged, stopSignal;

    public PoissonThread(ArrayWrapper wrapperV,
                         ArrayWrapper wrapperCopyV,
                         double[] F,
                         int start,
                         int end,
                         CyclicBarrier barrier,
                         AtomicBoolean allConverged,
                         AtomicBoolean stopSignal){

        this.wrapperV = wrapperV;
        this.wrapperCopyV = wrapperCopyV;
        this.F = F;
        this.start = start;
        this.end = end;
        this.barrier = barrier;
        this.allConverged = allConverged;
        this.stopSignal = stopSignal;
    }
```

```
@Override
public void run() {
    do {
        approximate(); //write approximation function to copyV

        // check if the function has converged in this thread's
            section
        boolean converged = hasConverged();
        if(!converged){
            // if this thread has not converged, signal others that we
            // are not ready to terminate
            allConverged.set(false);
        }

        // wait for all threads to agree to proceed
        try {
            barrier.await();
            // remember that this barrier uses runnable
            // (PoissonSolver.java @ line 59)
        } catch (BrokenBarrierException | InterruptedException e) {
            e.printStackTrace();
        }

        // if all threads have converged, terminate, otherwise
        // move to the next iteration
    } while(!stopSignal.get());
}

private void approximate(){
    double[] V = wrapperV.get();
    double[] copyV = wrapperCopyV.get();

    for(int i = start; i <= end; i++){
        copyV[i] = (V[i-1] + V[i+1] - F[i]) / 2.0D;
    }
}

private boolean hasConverged(){
    double[] V = wrapperV.get();
    double[] copyV = wrapperCopyV.get();

    for(int i = start; i <= end; i++){
        if(Math.abs(V[i] - copyV[i]) >= 0.001D * Math.abs(copyV[i])){
            return false;
        }
    }

    return true;
}
```

```
}
```

3.2 Data Evaluation Scripts

Listing 4: Bash scripts to run PoissonThreads 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 "threads,performance"
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 PoissonSolver $p)
      echo "$p,$performance"
    done
    echo ""
done
```

Listing 5: Python scripts to gather and plot results

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

def plotandsave(graph):
    plt.errorbar(graph['x'], graph['y'], yerr=graph['yerr'], ecolor='r')
    plt.xlabel('Number of threads')
    plt.ylabel('Performance (op/sec)')
    plt.xticks(range(33)[::2])
    plt.xlim(0,32)
```

```
    plt.title('Poisson Equation with Threads')
    plt.savefig('performance.png')
    plt.show()

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

    stats = {}
    l = 0
    graph = {'x':[], 'y':[], 'yerr':[]}

    for row in reader:
        if not row:
            graph['x'].append(l)
            graph['y'].append(statistics.mean(stats[l]))
            graph['yerr'].append(statistics.stdev(stats[l]))
        else:
            p = int(row[0])
            l = p

            if not stats.get(p):
                stats[p] = []

            stats[p].append(float(row[1]))

    plotandsave(graph)
```

Listing 6: Bash scripts to plot convergence over iterations

```
# This script gets the output of the PoissonSolver (which in this case
    needs to be
# the array at certain iterations) and plots them in a visual way

cd src
javac *.java && java PoissonSolver $1 > ../data.csv && cd .. && python
    plot.py data.csv
```

Listing 7: Python scripts to plot convergence over iterations

```
import sys
import matplotlib.pyplot as plt
import csv

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

    i = 0
    for row in reader:
```

```
    plt.xlim(0,4097)
    plt.plot(row, 'o')
    plt.savefig('{}.png'.format(i))
    i += 1
    plt.show()
```