[CPD-AP] AP 2021 TP2 Poisson

1 Abstract

We solve Poisson's equation in two dimensions using several variations on the Gauss-Seidel method. The Gauss-Seidel method is an iterative numerical method for solving systems of linear equations (the Poisson equation being a well-known example of such).
Multiple variations on the Gauss-Seidel algorithm were implemented in C and run on a high-performance computing platform (SeARCH at Universidade do Minho) and compared for performance.
Two techniques were applied to improve overall performance: Successive Over-relaxation (SOR), to increase the rate at which the algorithm converges on a solution, and "Red-Black" partitioning of the matrix cells combined with parallel programming to take full advantage of the multi-processor hardware.
Comparing these approaches, we were able to determine that using red-black improves the overall performance, but more work is needed.

2 Problem Description

2.1 Poisson's Equation for Stead-State Heat Distribution

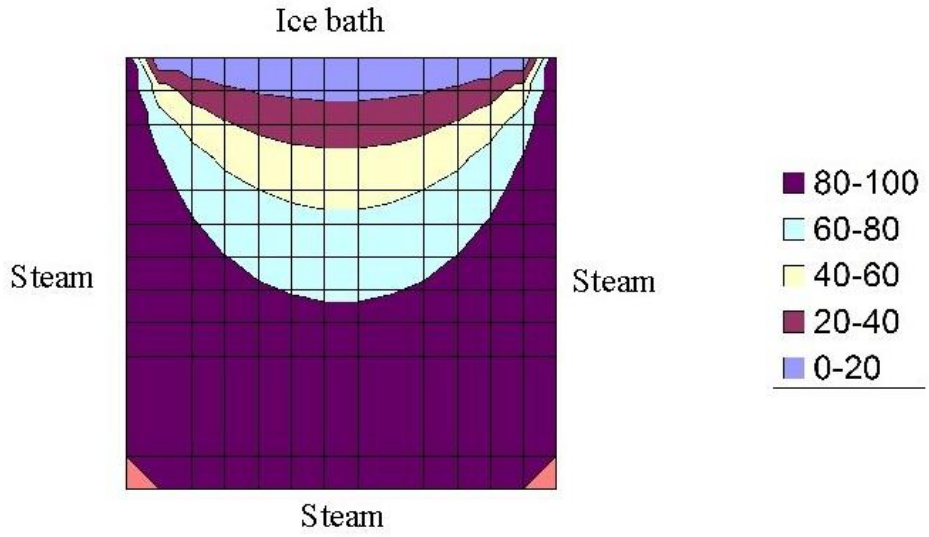

Figure 1 - Our Model: 2-D bath of liquid in a stead-state heat distribution

We model the example the distribution of heat across a bath of water. To simplify the problem, we treat the "bath" as a 2-dimensional square:

- Hot plates along the sides and bottom keep those edges at a constant temperature: 100 degrees.
- Across the top we use an ice bath to keep the top surface at precisely zero degrees.

We now *leave the bath to achieve a steady state.* The equation we wish to model is non-time dependent and represents the distribution of heat over the bath once it has reached a steady state. This is depicted in Figure 1 above.
Physics tells us that this heat distribution can be described by Poisson's equation:

$$\frac{\partial^2 u}{\partial x}(x,y)+\frac{\partial^2 u}{\partial y}(x,y)=g(x,y)$$

Equation 1 - Poisson's Equation in 2 dimensions

Poisson's equation is an example of an *Elliptical Partial Differential Equation*, which we simplify to Laplace's equation, by taking the function g = 0. Laplace's equation is an example of a *second-order partial differential equation* (PDE) and it is this sort of equation whose solution we will demonstration with numerical methods.

2.2 Gauss-Seidel

We further simplify the model by treating our *"bath"* as a grid of discrete points. We then apply variations of the Gauss-Seidel algorithm, starting with a rough approximation and refining it iteratively to arrive at the proper distribution – which is the solution we desire to Laplace's equation.
We also *invert* the tank such that the "ice" is at the bottom surface, to get a simple starting grid such as that shown in Figure 2 below.

	i=0	i=1	i=2	i=3	i=4	i=5	i=6
j=0	100	100	100	100	100	100	100
j=1	100	**50**	50	50	50	50	100
j=2	100	50	50	50	50	50	100
j=3	100	50	50	50	50	50	100
j=4	100	50	50	50	50	50	100
j=5	100	50	50	50	50	**50**	100
j=6	100	0	0	0	0	0	100

Figure 2 - Model Inverted "Bath" – Boundaries are constant

Notes:

- The blue shaded area in the center of Figure 2 is the actual area we will calculate. The boundaries are held constant.
- The grid cell indexing starts at the boundaries, even though these are not calculated
- The numbering starts at zero in each axis, because our algorithm is programmed in C and this is the most natural fit
- The first cell calculated will be the blue shaded cell at i=1, j=1 and the last i=5, j=5.
- The values *50* in the cells of the grid are merely a *guess* to use as a starting point for the algorithm: rather than choose a value at random, we take the mean of the two temperature extremes – this should result in a faster convergence to the real value.

The Gauss Seidel algorithm (GS) is generally an iterative numerical method for solving systems of linear equations. More specifically, it is well suited to solving partial differential equations such as that we find in our Poisson/Laplace equations. It is similar to the well-known Jacobi algorithm (not modelled in this experiment), with the added feature that Gauss-Seidel uses the results of each cell calculated immediately in

calculating the next cell, rather than doing the whole grid at once. Crucially, this makes for *faster convergence* in Gauss-Seidel.
Speed of convergence is crucial, since faster convergence means fewer iterations, which means better performance.
In our basic GS algorithm, we loop through the cells in the grid in a standard nested i,j loop and recalculated each cell as *the mean value of its four orthogonal neighbours.*

100	**top** 100	100	100	...
left 100	**(left + top + right + bottom) 4**	50 **right**	50	...
100	50 **bottom**	50	50	...
...	...	...	...	...

Figure 3 – Gauss-Seidel calculates each cell as the mean of its neighbours

Figure 3 above depicts this calculation in top right corner of the grid. Note that the white cells are used in the calculation, but are not technically part of the calculated grid. Hence we start our iteration at the dark yellow cell (which is at coordinate i=1, j=1 in our C implementation).
The pseudocode for this algorithm is shown in Figure 4 below.

```
Inputs: A, b
Output: φ

Choose an initial guess φ to the solution
repeat until convergence
    for i from 1 until n do
        σ ← 0
        for j from 1 until n do
            if j ≠ i then
                σ ← σ + a_ij φ_j
            end if
        end (j-loop)
        φ_i ← 1/a_ii (b_i − σ)
    end (i-loop)
    check if convergence is reached
end (repeat)
```

Figure 4 - Basic Gauss-Seidel Algorithm (Wikipedia contributors, 2021)

(In our implementation of the algorithm, we do not exclude the diagonals as in the pseudcode quoted) and we can represent our inner calculation more simply as that depicted in Equation 2.

$$w_{i,j} = \frac{\left(w_{i-1,j} + w_{i,j-1} + w_{i+1,j} + w_{i,j+1}\right)}{4}$$

Equation 2 – Gauss-Seidel 2D calculation of grid cell

More simply (but less mathematically) put, the value of the cell at i, j is

$$w_{i,j} = \text{mean}(\text{left} + \text{top} + \text{right} + \text{bottom})$$

Convergence of the algorithm is determined by comparing the grid at the end of each iteration with its previous state – that is, performing a matrix subtraction – and finding the largest difference between cell and its prior state. This difference (*diff*) is compared against a *tolerance*, which defaults to the real number 0.001 such that if the difference is less than the tolerance, convergence is considered achieved and the iterations cease.

2.3 Successive Over-relaxation

The plain GS algorithm can tend to be slow to converge. To improve on this, we introduce Successive-Over-relaxation (SOR) (Ruan, 1990). In the SOR algorithm, we apply the same mean-of-neighbours calculation, but apply a *relaxation factor* and take the *weighted average* of the mean-of-neighbours and the *prior* value. To simplify this algorithm, we'll start with *m* as the *mean* from Equation 2:

$$m = \text{mean}(\text{left} + \text{top} + \text{right} + \text{bottom})$$

and *p* as the relaxation factor. We get the new calculation for the cell at i, j:

$$w_{i,j}^{k} = (1 - p)w_{i,j}^{k-1} + p * m$$

Equation 3 – SOR uses the weighted average of the last value and the mean

It should also be noted that the simple Gauss-Seidel algorithm from h2.2 is a special case of the SOR algorithm, where *p = 1* (i.e. the calculated mean gets the full weight)
The value of *p* can be provided by command line argument in our program, but defaults to a value between 0 and 2 based on the size (n) of the matrix generated. The formula we use by default is:

$$p = \frac{2}{1 + sin\left(\frac{\pi}{n-1}\right)}$$

Equation 4 – default relaxation factor based on size n

By taking the weighted average, with our relaxation factor *p* as the weight, we converge more rapidly on a solution. Faster convergence means better performance for the same result, which is our goal here.

2.4 Parallelization with Red-Black

It is clear from close examination of section 2.2 above that the basic Gauss-Seidel algorithm does not lend itself well to parallel programming techniques. This is because of the way in which the calculation of each cell depends on the cell before it.

	i=0	i=1	i=2	i=3	i=4	i=5	i=6
j=0	100	100	100	100	100	100	100
j=1	100	50	50	50	50	50	100
j=2	100	50	50	50	50	50	100
j=3	100	50	50	50	50	50	100
j=4	100	50	50	50	50	50	100

j=5	100	50	50	50	50	**50**	100
j=6	100	0	0	0	0	0	100

Figure 5 - overlapping updates

In Figure 5 above, the *sequential* algorithm will update the cell **`i=1,j=1`** and then the cell at **`i=2,j=1`**. If we try to parallelize the algorithm, such that both cells are updated simulateneouly, we create a race condition, since each cell's value depends on the outcome of the caclulation for the other.
One solution for this dependency problem is to break the grid up into alternating "red-black" squares, such that their direct neighbours – the input for their calculations on each iteration – do *not* overlap.
This breaks the grid into two overlapping grids: a red grid and a black grid, such that each grid has a set of completely independent cells, but one grid is highly dependent the other.

	i=0	i=1	i=2	i=3	i=4	i=5	i=6
j=0	100	100	100	100	100	100	100
j=1	100	**50**	50	50	50	50	100
j=2	100	50	50	50	50	50	100
j=3	100	50	50	50	50	50	100
j=4	100	50	50	50	50	50	100
j=5	100	50	50	50	50	**50**	100
j=6	100	0	0	0	0	0	100

Figure 6 - Red-Black division of grid cells

This allows us to employ parallelization – especially simple parallelization of the type provided by OpenMP with its `omp for` loops – on each of two sub-grids. The implementation of this technique is described more in section 4.1.4 below.

3 Testbed Environment

Our hardware test environment was prepared by requesting time on a specific queue on the SeARCH cluster. The following command is issued:

```
qsub -I -qmei -lnodes=1 -lwalltime=1:30:00
```

This subscribes to a single node of a server in the MEI queues. In our case, we tested our program in the 652 server. This server is a dual-socket Intel Xeon ES-2670v2 **20 core** processors running at up to 2.50GHz with 64GB of RAM.

Excerpt from lscpu:

```
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
CPU socket(s):         2
Vendor ID:             GenuineIntel
CPU MHz:               2500.000
```

```
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
```

4 Program

4.1 Overview

The test program was written in C, with a main program that loops for the given number of processes, and then selects one or all of the algorithms (based on command line arguments) and gathers results in a metrics object.

4.1.1 Algorithms

The algorithms are combined in one file `poisson.c` and selected by use of an enum called `algo_t`:

```c
enum algo_t {
    gs = 0,             // Gauss Seidel
    gsrb = 1,           // Gauss Seidel with red-black
    sorrb = 2,          // Successive Over-relaxation with red-black
    gsrb_parallel = 3,  // Gauss Seidel with OpenMP parallel processing over red-black
    sorrb_parallel = 4, // SOR Seidel with OpenMP parallel processing over red-black
    all = 5,            // All 4 algorithms in a loop
    compare = 6         // Special: compares gsrb and gsrb_parallel to help tweak OMP
};
```

Listing 1 – enum for selecting algorithm

Note that:

- The `_parallel` enum values are not really different algorithms: this is simply an easy way to allow us to experiment between a pure red-black algorithm, and the OpenMP parallelized equivalent
- The `compare` enum is only for use during development: it reports the speedup gained by the OpenMP-parallelized algorithm over the plain GS Red-Black, to allow us to *tweak* the OMP code.

  Sample output of the compare algorithm:

  ```
  > bin/poisson --warn --algo compare --size 200
  Sequential:      2.046
    Parallel:      4.362
  Parallel was 113.18% SLOWER! Overhead unacceptable with 16 threads. Fix OMP code
  ```

  It is very easy to write non-optimal OMP code such that the overhead of managing threads costs more than the benefit. The output above shows just such a case, where more tweaking of the OMP was needed.

4.1.2 Gauss-Seidel

Listing 2 below is an excerpt of the main program showing the GS algorithm:

```c
void poisson_algo(int n, double w[n][n], double tolerance,
```

```
                double relaxation, bool red_black, bool parallel, struct metrics *metrics)
{
    double diff;
    diff = tolerance + 1; // initial diff must be > tolerance
    int iterations = 0;
    double u[n][n]; // matrix from prior iteration
    double d[n][n]; // matrix to hold the difference between u and w

    while (diff > tolerance) {
        copy_matrix(n, w, u);
        if (red_black) {
            if (parallel) {
               red_black_parallel(n, w, relaxation);
            }
            else {
                red_black_sequential(n, w, relaxation);
            }
        }
        else {
            for (int i = 1; i < n - 1; i++) {
                for (int j = 1; j < n - 1; j++) {
                    calculate_cell(n, w, i, j, relaxation);
                }
            }
        }
        iterations ++;
        // maximum cell value in the matrix difference between this iteration and the last
        // if the max cell difference is less than the tolerance, we are done
        abs_diff(n, w, u, d); // abs diff: d = w - u
        diff =  max_cell(n, d); // diff is the largest difference between any corresponding cells
     }
}
```

Listing 2 – Gauss-Seidel implementation

The implementation for each cell calculation is in `calculate_cell`, which varies across the simple and SOR calculations, as seen in Listing 3:

```
void calculate_cell(int n, double w[n][n], int i, int j, double relaxation)
{
    double above = w[i-1][j];
    double left = w[i][j-1];
    double right = w[i][j+1];
    double below = w[i+1][j];
    double mean = (above + left + right + below) / 4.0f;
    double new_value = mean;
    if (relaxation >= 0) { // <= 0 means no relaxation, 0 means no change!
        // v = relaxation-factor * the gauss-seidel value + 1-relaxation * the old value
        // e.g. if relaxation = 0.5 we only take half the mean and half the old value
        new_value = ((1.0 - relaxation) * w[i][j]) + (relaxation * mean);
    }
    w[i][j] = new_value;
```

Listing 3 – Gauss-Seidel cell calculation – with plain and SOR versions

4.1.3 Successive Overrelaxation

The SOR implementation is simply added as a relaxation parameter in the main `calculate_cell` function: if the relaxation is -1, no SOR is applied, otherwise it is used to obtain the weighted average of the calculated mean and the prior value. (Note that technically we could simply use 1.0 as the non-SOR value, since the SOR calculation collapses to the plain GS with that value, but the program performs better if we skip that calculation.

4.1.4 Red-Black

The red-black calculation variations can be seen called from Listing 2 above. The functions they call vary depending on whether we are using OpenMP parallelization or not. Clearly the most relevant code is the one using OpenMP, which is shown in the listing below:

```
void red_black_parallel(int n, double w[n][n], double relaxation) {
// RED cycle
    int i, j;
#pragma omp for
        for (i = 1; i < n - 1; i++) {
            // red starts at (i+1 modulo 2), so i=1:j=1+2%2=1;  i=2:j=1+3%2=2
            for (j = 1 + ((i + 1) % 2); j < n - 1; j += 2) {
                calculate_cell(n, w, i, j, relaxation);
            }
        }
#pragma omp barrier
    }
    {
// BLACK
#pragma omp for
        for (i = 1; i < n - 1; i++) {
            // black starts at 1 + (i modulo 2), so i=1:j=1+1%2=2;  i=2:j=1+3%2=2
            for (j = 1 + (i % 2); j < n - 1; j += 2) {
                calculate_cell(n, w, i, j, relaxation);
            }
        }
#pragma omp barrier
    }
}
```

Listing 4 – Red-Black algorithm with Parallelization using OpenMP

Listing 4 depicts the main red-black implementation, from which we can note:

- The red implementation is completed entirely before the black implementation is begun: this ensures no overlap.
- The starting point for the inner j loop varies for odd (red) and even (black) rows, hence `i + 1 modulo 2` is performed for red and `i modulo 2` for black.
- In addition to the separation in code, the `omp barrier` directive ensures that all threads are finished working on the red grid cells before the black section begins.
- Given that the sections do not overlap, we are able to apply OpenMP in the form of `omp for` loops, to leverage system threads to share the performance.
- No collapsing is done on the nested for loops, since the size of the outer loop (i) is much greater than the number of threads, hence no benefit is obtained by collapsing further.

4.1.5 Performance Improvements: Speeding up the Diff Bottleneck

Initial runs showed very poor performance. Examination and debug showed that the calculation of the difference at the end of each iteration was a bottleneck.

```
  abs_diff(n, w, u, d);  // abs diff: d = w - u
  diff = max_cell(n, d); // diff is the largest difference between any corresponding cells
```

Listing 5 – Slow difference calculation

The diff calculation in Listing 9 uses full matrix subtraction and scanning to find the maximum value in the difference between the matrices. This was modeled on the matlab routine which did as here:

```
    DIFF=max(max(abs(w-u)));
```

Listing 6 – MATLAB difference calculation

Note that the MATLAB calculation is not only much more concise, but is likely optimized by matlab. Our copy of that implementation, on the other hand, is slow.
To speed this up, we take advantage of the fact that we are already looping to calculate the new cell values, so we calculate the max difference as we go along, rather than all at the end.
In the OpenMP implementation, this allows us to take advantage of the `reduction(max)` feature.

```
    int i, j;
    double diff = MAXFLOAT;
    double max_diff = 0;
#pragma omp parallel for schedule(runtime) shared(n, w, relaxation) private(i,j, diff) default(none)
reduction(max:max_diff)
    for (i = 1; i < n - 1; i++) {
        // red starts at (i+1 modulo 2), so i=1:j=1+2%2=1;  i=2:j=1+3%2=2
        for (j = 1 + ((i + 1) % 2); j < n - 1; j += 2) {
            diff = calculate_cell(n, w, i, j, relaxation);
            max_diff = max_diff > diff ? max_diff : diff;
        }
```

Listing 7 – Improved difference calculation

The highlighted lines in Listing 7 show the calculation of the difference along the way.
With these improvements, the parallel version of the program – using OpenMP – is finally faster than the sequential – as expected!

```
[pg42819@compute-652-2 scripts]$ ./compare_parallel.sh -f
running with sequential then parallel GSRB with size 400 to evaluate OMP code
Sequential:     19.449
  Parallel:     16.463
Parallel with 40 threads was faster by 18.14%
```

Listing 8 – Improved results!

4.1.6 Command line arguments

The command line allows each or all of the algorithms to be run, and the size of the matrix to be configured as well as the tolerance.

In fact, full configuration is provided: the starting temperatures can be set, and the SOR relaxation factor can be overridden from its default.
Listing 9 below displays the help output, which describes all of these options.
Note that the timing results can be output to a csv file to allow us to build up a result set of over many runs.

```
❯ bin/poisson -h
Usage: poisson -n <size> [options]
Options include:
    -n --size NUM number of cells to a side of the square bath
    -a --algo ALGO name of algorthm; one of: gs, gsrb, sorrb, gsrb_parallel or sorrb_parallel
    -t --tolerance NUM max difference between iterations before stopping
    -r --relaxation NUM custom relaxation for SOR between 0 and 2 (default: 2/(1+sin(pi/(n-1)) )
    -p --processes NUM number of iterations to perform
    --wall-temp TEMPERATURE initial temperature of the walls and bottom (top is always zero; default: 100.0)
    --inner-temp TEMPERATURE initial temperature of inner cells (default: 50)
    -m METRICS.CSV append metrics to this CSV file (creates it if it does not exist)
    -e --test run the algo with a test matrix of 4x4 and 1 process with a known outcome (-n and -p ignored)
    --info for info level messages
    --verbose for extra detail messages
    --warn to suppress all but warning and error messages
    --error to suppress all error messages
    --debug for debug level messages
    --trace for very fine grained debug messages./TP1/bin/rooms_sa -h
```

Listing 9 – help output for the command line

4.1.7 Validating algorithms

The `--test` option is provided for validating an algorithm for correctness. It sets up a constant initial assignment of a 4x4 tank and compares against an expected outcome within tolerance. By running with

`--test --trace` you can see the individual steps taken to achieve the results. You can also see a warning if the test fails.
For more details see the attached codebase in GitHub: https://github.com/FranciscoRosa11/AP.

4.2 Compilation and Runtime

The program is compiled with GCC 5.3.0 on the SEARCH platform.

```
gcc -O3 -std=c99 -g  -fopenmp  -lm -o bin/poisson src/main.c src/poisson.c src/config.c src/matrix_support.c
```

Bash scripts are used to run all versions of the program with varying values for the number of processes, the tolerance, relaxation parameter and so forth.
The number of threads used by OpenMP is varied between 10 and 40 with the use of the OMP_NUM_THREADS environment variable. (Only these values were chosen, as the number of threads did not have a very significant effect on the results)

5 Results

Test results were collated by generating output into a csv file, resulting in 97 data rows, given all of the permutations of the different inputs.
Table 1 below summarizes the best times and required iterations for each of the algorithms, with the size of the grid varying from 5000 to 25000 cells.

Size / Algo	Best Time (seconds)	Min Iterations
5000		
GS	87	2169
GS-RB	50	2186
GS-RB-OMP	2340	2186
SOR-RB	3	130
SOR-RB-OMP	165	2186
10000		
GS	297	3660
GS-RB	165	3684
GS-RB-OMP	473	3684
SOR-RB	9	180
SOR-RB-OMP	344	3684
15000		
GS	594	4879
GS-RB	323	4908
GS-RB-OMP	5025	4908
SOR-RB	17	214
SOR-RB-OMP	556	4908
20000		
GS	988	5959
GS-RB	535	5992
GS-RB-OMP	811	5992
SOR-RB	26	245
SOR-RB-OMP	756	5992
25000		
GS	1459	6928
GS-RB	794	6965
GS-RB-OMP	1078	6965
SOR-RB	37	274
SOR-RB-OMP	1003	6965

Table 1 - Summary results over all algorithms (numbers are mean values over many runs)

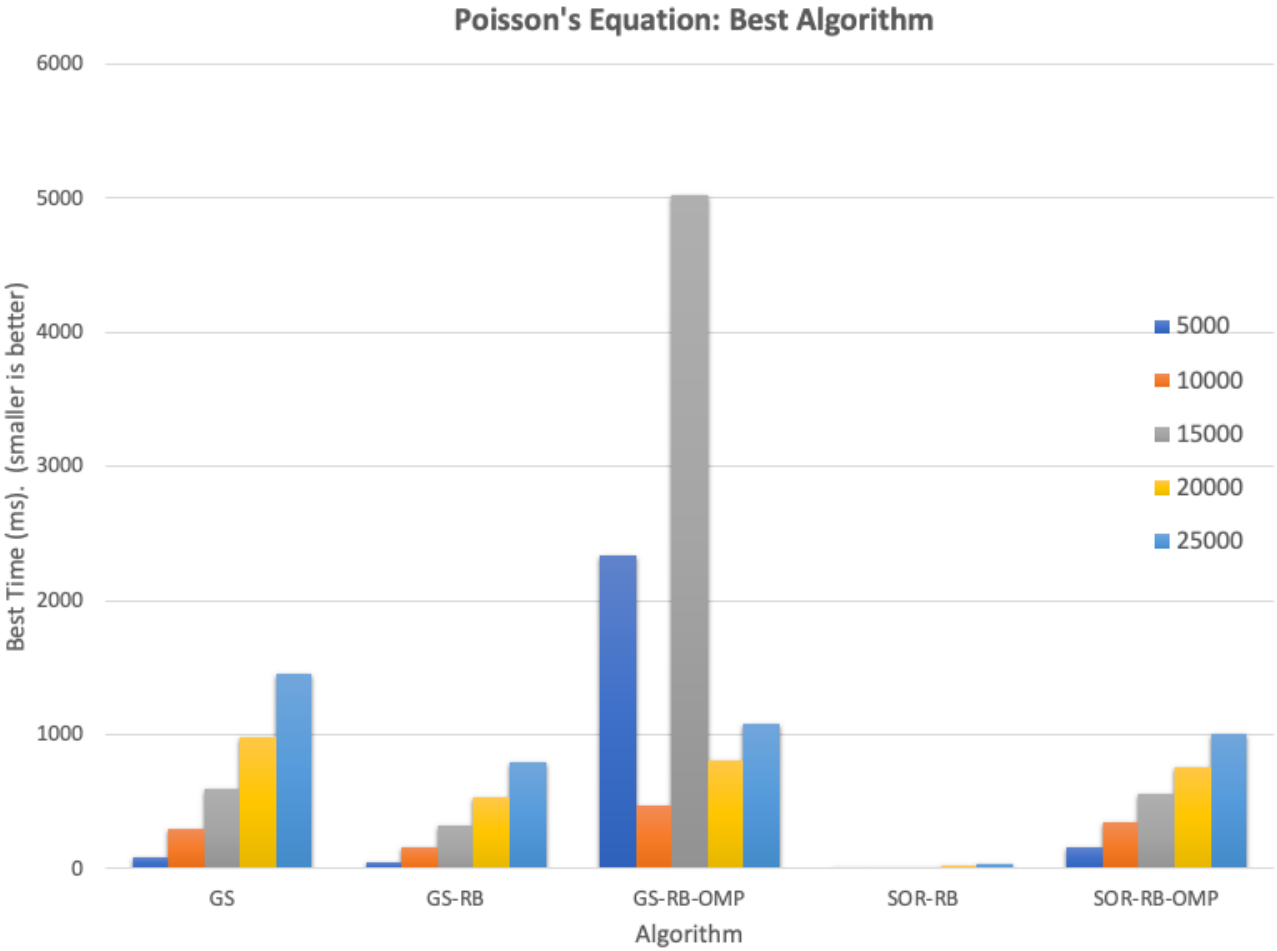

Figure 7 - Overall comparison of the different algorithms – smaller bars are better

The overall comparison in Figure 7 above compares the various algorithms, showing the best time staggered across the different sizes tested.

It is clear from this that the SOR-RB performed better.

Meanwhile in Figure 8 below, we can see that the time take grows almost linearly, as we would expect, with the grid size. What we did *not* expect is that the GS red-black with OMP would be nearly identical to the SOR red-black *without* OMP. It is likely some external artifact: such as an external load on the server at that point in time (since the three algorithms are run in succession, in the same process, for a given grid size)

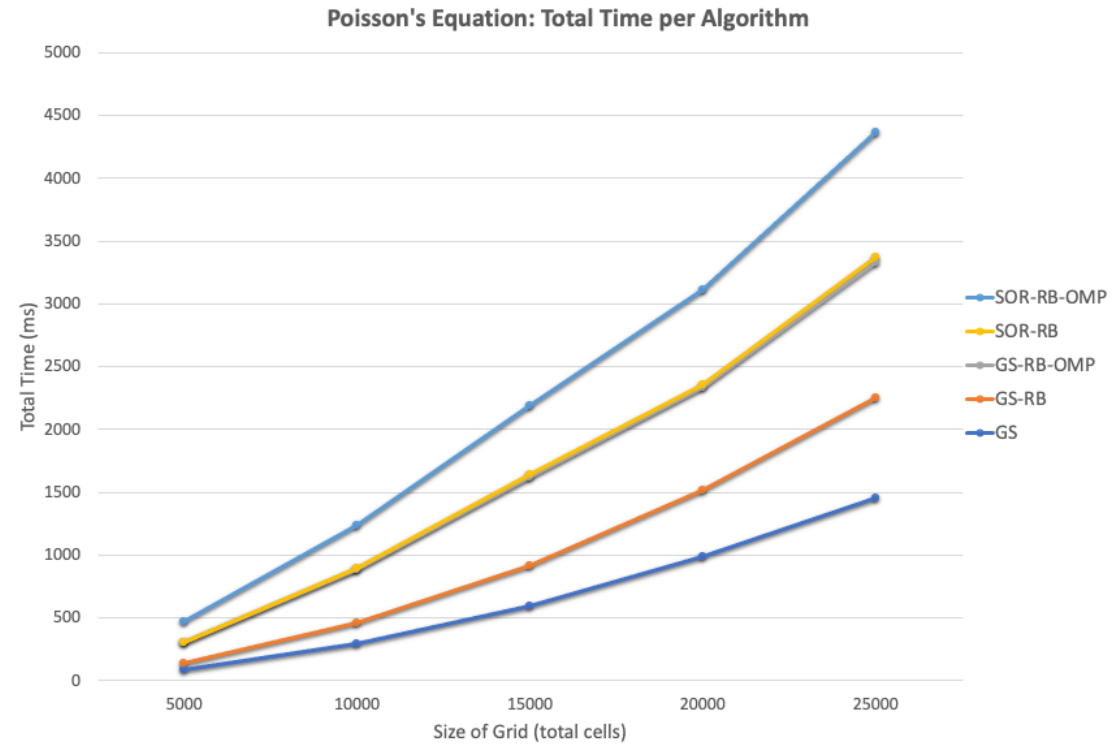

Figure 8 – Total time for each algorithm by grid size

The last graph in Figure 9 below shows nothing surprising: the number of iterations needed increases linearly as the grid size.

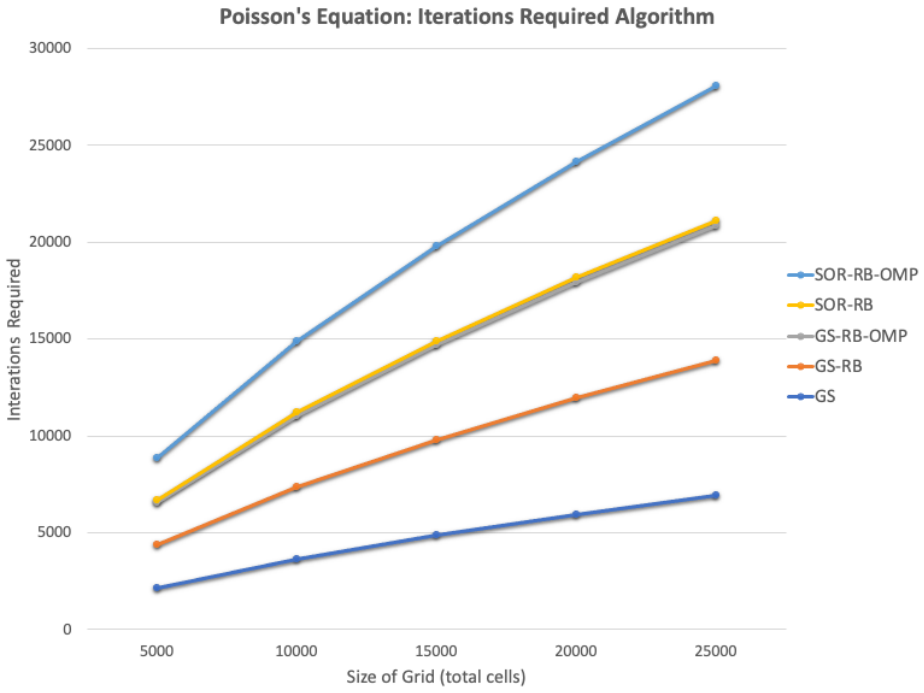

Figure 9 - Required Iterations for Grid Size

6 Conclusions and further research

The conclusions of these experiments are born out in the graphs in the previous section, and the experimentation documented along the way:

- Successive Overrelaxation gives better performance given that it converges faster – the results show that this expected result was achieved in Figure 7 above.
- The use of red-black allows us to apply OpenMP to parallelize the workload by avoiding dependencies.
- The OMP implementation needs more work to get the maximum benefit.

Next steps:

- Try larger data sizes – with hours long runs to even out the artifacts
- Use blocking within the nested loops to take advantage of cache performance (the large matrix sizes we are using for the double precision numbers will overflow L1 caches quickly)

7 Works Cited

Ruan, S. (1990). Successive overrelaxation iteration for the stability of large scale systems. *Journal of Mathematical Analysis and Applications, 146*(2), 389-396.

Wikipedia contributors. (2021, February 13). *Gauss–Seidel method*. Retrieved from Wikipedia: https://en.wikipedia.org/w/index.php?title=Gauss%E2%80%93Seidel_method&oldid=1006613059