UNIVERSITY of CALIFORNIA

SANTA CRUZ

SOLVING BOUNDARY VALUE PROBLEMS IN PHYSICS EFFICIENTLY USING COMPUTERS

A thesis submitted in partial satisfaction
of the requirements for the degree of

BACHELOR OF SCIENCE

in

PHYSICS

by

Daniel A. Bittman

May 25, 2016

The thesis of Daniel A. Bittman is approved by:

Professor Jason Nielsen
Thesis Adviser

Professor David P. Belanger
Thesis Coordinator

Professor Robert Johnson
Chair, Department of Physics

Copyright and Software License

Dedication

To my loving parents, Steve Bittman and Ellen Klima,
for their unending support and unwavering belief in
my ability to succeed.

Also for Frankie.

```
 \    /\
  )  ( ')
 (  /  )
  \(__)|
```

Acknowledgements

A huge thank you to my thesis adviser, Jason Nielsen, for providing the opportunity to work on this interesting project, and for the many interesting conversations over the quarters past.

The Storage Systems Research Center at UC Santa Cruz for some of their machines in order to test the performance of this system.

Contents

List of Figures

List of Tables

List of Source Code Listings

Abstract

There are many problems in physics which involve solving a set of equations that define behavior inside a region. One such set of problems, among many, are those of electrostatics inside a region given some boundary conditions. Such problems, along with other problems, are commonly found in simple and analytically solvable forms in undergraduate physics classes. Unfortunately, these problems can quickly become intractable to solve analytically as their complexity increases. Complex boundary conditions in strangely shaped regions can easy become impossibly difficult to solve. The solution can instead be modeled on a computer in order to compute a numerical solution in place of an analytical one.

I have written a software package with both a C library capable of solving such a given problem, and a front-end Python program which can parse a human-readable configuration file that describes such a boundary value problem. The Python code sets up the correct initial state such that the C library can solve it, after which the Python program displays the final result and calculates statistics about the solving process. I have utilized effective programming techniques designed for current generation 64-bit x86 processors in order to improve the performance of the program, and have implemented a multi-threaded version of the solving program to explore the use of multi-core processors in solving these problems.

I found that the program correctly matched an analytical solution to a simple test problem, and that it correctly predicted the $1/r^2$ potential found from an electric dipole. I was able to optimize the program so that it efficiently found these results, and I was able to successfully leverage multi-threading to significantly increase the performance, albeit with a degradation in simulation accuracy.

1 Introduction

Boundary value problems are problems defined by a partial differential equation and a set of constraints on the solution, and they are found in many areas of physics. These include problems such as finding the temperature along a rod that has a given temperature on either end, determining the electric potential inside a region given the potential around the boundary, or determining the flow of an incompressible fluid. As the complexity of these problems increases, however, the feasibility of solving them analytically quickly decreases. Instead, we have the option to solve them with iterative methods using computers. These types of problems are well-suited for computers because they are easily parallelizable, can be discretized easily, and require a lot of repetitive calculation [8]. I have developed an implementation of such a solver for electrostatics utilizing the parallelizability of the problem and designed it to be efficient by taking advantage of the design of modern CPUs. It can produce a solution to given problem given a description of a problem to solve.

For this I have two objectives: ensuring that the simulation produces the correct result, and having the program solve the problem quickly. The first of these can be done by comparing the result to a known solution and defining a metric for determining how close it is. This should provide an accurate way to measure how correct the simulation is, allowing me to compare different methods and ensure that they are all correct. The second of these objectives can be realized via two methods. Firstly, use an algorithm that has a reasonably small computational complexity, and secondly, implement the algorithm in a way that is efficient for modern computers.

1.1 Poisson's Equation

Although many differential equations are typically used in boundary value problems, I will limit the focus to the Poisson equation. Named after Siméon Denis Poisson, the Poisson equation is used in a number of typical boundary value problems, including electrostatics. In 2-dimensional Euclidean space it is,

$$\nabla^2 u = f,$$

where f is a real-valued function inside the space and u is the steady-state solution for the potential in the space[1]. The problems that we are considering are called boundary value problems because their solutions are dependent not only on the value of f, but also on the properties of the region and its boundaries. Formally, the problems to solve are given with f defined and with a region Ω which has defined boundaries $\partial\Omega$. These boundary conditions, along with f, are the primary inputs that determine what solution the program will give.

1.2 Discretizing Poisson's Equation

Since we are solving these problems using a computer, we must transform the problem definition into one which a computer can actually calculate. Computers do not do well with continuous functions and real numbers. The process of transforming a problem or equation into one which a computer can handle is called discretization.

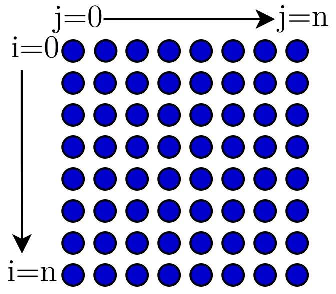


Figure 1: Visualization of grid, showing indices. The choice of which axis is i and which is j is arbitrary.

Consider a grid of size n by n as in Fig. 1[1], with values represented by $u_{i,j}$. We want to apply the Poisson equation to this u as follows:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} = f,$$

however, u is not continuous, so the partial derivative for x becomes

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2},$$

at a grid point (i, j), where $h = 1/n$ is the width and height of a grid cell. The partial derivative for y is similar, using j instead of i. This gives the full discrete Poisson equation as[5][9],

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = f_{i,j}.$$

[1]In general, it is not difficult to allow a rectangular grid, but for simplicity we will require the grid to be square.

Since $u_{i,j}$ is what we are interested in, this is more useful after some rearranging:

$$h^2 f_{i,j} = -4u_{i,j} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1},$$

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}). \tag{1}$$

It can be seen from the top equation that this is now a linear algebra problem, requiring a method of solving n^2 dependent linear equations. This means that we can apply one of several standard iterative methods for solving the problem. This is an excellent application of using a computer to numerically solve a problem which is difficult to solve analytically.

1.3 Jacobi Iteration

The Jacobi iteration method is perhaps the simplest method of iteratively solving this linear algebra problem. The procedure involves treating each equation as independent, and solving iteratively. This turns Eq. 1 into[5]

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k} - h^2 f_{i,j}), \tag{2}$$

where k is the iteration number. This is saying that to calculate u everywhere, we must start with a guess for u, and then iteratively solve each element in u, which depends on its four nearest neighbors. One immediate unfortunate reality with

Jacobi iteration presents itself: we must store a secondary copy of the grid because each calculation of $u_{i,j}^{k+1}$ strictly requires values from the previous iteration. Since we iterate over the grid in order to solve it, we cannot write the new value for the $k+1$ iteration into that element of u, since we would be overwriting a value that is needed later. This unfortunately but necessarily increases the space required by $O(n^2)$.

1.4 Boundaries

The above equation does not yet include any consideration of boundary conditions. We will consider Dirichlet boundary conditions as the primary type of boundary conditions we will support. Dirichlet boundary conditions are defined such that the solution ϕ of a partial differential equation in a region Ω has a value $\phi(\mathbf{r}) = y(\mathbf{r})\ \forall \mathbf{r} \in \partial\Omega$. For this case, this means that if a Dirichlet boundary value condition is specified for a particular grid cell (x, y), then its value is simply fixed to the value of the Dirichlet condition for that cell, and that cell does not need to be recalculated on successive iterations.

Another consideration is that the grid is of finite size. This means that when calculating $u_{i=0,j}$, for example, we cannot incorporate $u_{i-1,j}$ as Eq. 2 would dictate. This is because that would require the -1^{st} row of the matrix u, which does not exist. A common method of dealing with this problem is to "wrap around" to the other side – essentially considering the region Ω to the surface of a sphere, or to consider the space to be periodic. Because of the nature of the problems that I will be considering, I have chosen to consider an implicit Dirichlet boundary condition of zero everywhere outside of Ω, thus effectively limiting the calculation to be inside the region while clamping any values "outside" of Ω to zero. This is easy to implement and visualize as well: any value of $u_{i,j}$ for $i \geq n$, $i < 0$, $j \geq n$, or $j < 0$ is simply zero, always.

1.5 Successive Over-Relaxation

An alternate approach of iteratively solving this problem numerically is called successive over-relaxation (or SOR), and it has advantages over simple Jacobi iteration. It works by reusing previous calculations for the next iteration when calculating a given cell. It transforms Eq. 2 into,

$$u_{i,j}^{k+1} = (1-\omega)u_{i,j}^{k} + \frac{\omega}{4}\left(u_{i-1,j}^{k+1} + u_{i+1,j}^{k} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k} - h^2 f_{i,j}\right), \tag{3}$$

where ω is a tunable parameter called the *over-relaxation parameter*. For a square lattice, a value of

$$\omega = \frac{2}{1+\frac{\pi}{n}}$$

has the best rate of convergence [5]. There are two terms on the right-hand-side of Eq. 3 which refer to the $k+1$ iteration. This approach uses the new values for the cells that have already been updated on this iteration in order to speed up the convergence.

Successive over-relaxation has a significant advantage over Jacobi iteration, in that it will result in a significantly faster convergence. These two iterative solutions require iterating over every cell in the grid a number of times. However, the number of iterations needed to converge is not constant – it is dependent on the size of the grid. For Jacobi iteration, the number of iterations required is proportional to n^2, where n is the length of one side of the grid, thus making the number of iterations $O(n^2)$ and so the total algorithm run in $O(n^4)$. SOR beats this by having the number of iterations be proportional to n, making SOR an $O(n^3)$ algorithm [5]. This means that SOR is not only significantly faster than Jacobi iteration for large enough n, the improvement for SOR over Jacobi iteration increases as the grid size increases. Furthermore, SOR has an advantage in that it requires less extra space for the computation because it is allowed to reuse values that have been calculated already in the same iteration, removing the requirement to store old values of cells like in Jacobi iteration. For this reason I predicted that, while they give the same results eventually, SOR would be a much faster method than Jacobi iteration in the solving program.

1.6 Performance Considerations

Because the grid can be large, it would be useful to section of parts of the grid and work on them simultaneously. To do this, we can leverage multi-threading. If we allow distant regions to be operated on by different threads I believe it to be unlikely that this will cause interations, at least while the number of threads is small compared to the grid size. To explore this, I have implemented a multi-threading approach to solving the problem with the understanding that it could cause erroneous behavior near where two threads are operating on nearby cells. However, I believe these effects will be small and will average out, so the iterative methods will still work correctly within the limit of these interaction regions being few in number.

While operations on a grid of values are parallelizable, there is a limit to the effectiveness of multi-threading the algorithm. For example, an average modern processor with 4 cores each with a 256KB cache and a shared cache of 8MB. Each core can only operate on a grid of 256 by 256 32-bit values before exhausting its cache. A performance drop will occur if the grid size per core exceeds the core's cache size. For example, should the grid size exceed 1440 by 1440 32-bit values, the shared 8MB cache will be filled, resulting in a massive performance penalty. For this reason, naïvely splitting the grid into 4 areas and multi-threading the iteration may result in sub-par performance. However, like most performance questions, this is difficult to predict.

There are many other aspects of processor design that come into play when writing a program such as this. Multi-threaded synchronization aside, there are many details of caching that need to be considered. Of course, optimizing a program is just like any experiment in that it is absolutely vital to apply normal scientific practices when doing such work. Many make the mistake of simply writing what they believe to be a more optimized version of a program without measuring the resulting performance and comparing it to the previous version.

I would expect to see the performance enhanced by the use of multi-threading, but insignificantly due to the large amount of memory accesses made by the program throughout solving a given problem. I expect to see a large performance improvement between a naïve implementation of the program and one where memory access patterns have been tuned.

Using this information, I have written a program which supports both Jacobi iteration and successive over-relaxation, allowing a varying number of threads to be used, and can calculate statistics about the solution it generates. It can be given an arbitrary input file to define the parameters, boundary conditions, and initial conditions for the region it is simulating. This allows me to simulate known problems and compare them to known solutions in order to verify correctness, and then I will be able to measure the performance characteristics of different implementations and solution methods with different parameters and compare them to see which are faster.

2 Methods

The solving program is written in C[2] in order to leverage the full performance of modern computers and is implemented as a shared library (on UNIX) such that it can be linked with another program which sets up the initial conditions before calling the solving function. The C code that does the solving is hereafter referred to as the engine, while any code that interfaces with the engine and provides a usable user interface is referred to as the frontend.

I have written an example frontend in Python, which is capable of parsing a configuration file which describes the values of f, the initial boundary conditions, the grid size, and a number of other details about a given desired setup (hereafter collectively referred to as a configuration). I have written several example configurations which I have used to confirm the correctness of the solving program. For a more detailed technical description of the implementation, see Appendix C.

2.1 Configurations and Mapping to Electrostatics

These configuration files are designed to be easy to write. An example configuration file is the following:

[2]Specifically the 2011 revision to the C programming language. This means that a new enough C compiler, one which supports C11, is needed in order to compile the program.

```
        gridsize 300
        dirichlet 1,1 1,grid.len = math.sin(y * math.pi / grid.len)
        cell 10,10 initial -100
```

The first line sets up a grid of size 300 by 300. The second line specifies that a Dirichlet boundary condition exists from cell $(1,1)$ to $(1,300)$ which takes on a value given by the expression at the end of that line. Any valid Python expression may be put here, allowing access to complex mathematical expressions. Additionally, the start and end points do not have to define a line on a single axis; the program will do a linear interpolation to create a straight line from the start point to the end point. The third line sets the initial value of the cell $(10,10)$ to a value of -100. This is the command used to define $f_{i,j}$ in Eq. 1. An arbitrary number of these statements may be made inside a configuration file, allowing one to specify the value of f everywhere if need be, or create Dirichlet boundary conditions of arbitrary shapes.

The available commands for a configuration file map to various initial and boundary conditions for an electrostatics problem in a straightforward way: the command to specify a Dirichlet boundary condition is like creating a boundary of fixed potential. The command to specify the initial value of cells is like specifying an initial charge distribution inside the region.

2.2 Verifying Correctness

The clearest way to verify that the solver is working correctly is to test it out on a number of different configurations for which the correct solution is known. My first choice here was a simple electrostatics problem straight out of a textbook [6]:

> Consider a square region with the bottom left corner at the origin with each side of length L. Let the potential along each side of the region be fixed at 0, except for the bottom side, which has a potential $V(x) = V_0 \sin(x\pi/L)$. What is the potential V inside the region?

This problem is equivalent solving Laplace's equation (which is Poisson's equation with $f = 0$), given some initial boundary conditions (hereafter referred to as "the $\sin(x)$ potential"). It can be solved analytically using a standard separation of variables trick [1]. Assume

$$\phi = X(x)Y(y),$$

then

$$X''(x)/X(x) = -Y''(y)/Y(y) = -\lambda,$$

and by applying the boundary conditions and using several commonly used mathematical physics techniques, one comes to the solution[6]

$$V(x, y) = \frac{\sinh(y\pi/L)}{\sinh(\pi/L)} \sin(x\pi/L).$$

In order to compare the result of my program with the analytical solution, I added a feature to the Python frontend to accept a pre-made file containing a precalculated grid of the correct output by using the analytic solution. The frontend then reads this file and compares it to the result of the solver.

Since there is now a problem which has an analytical solution, I was able to use that problem to describe a metric for correctness of the simulation. I wrote a Python script that generated a 2-D grid of values based on the analytical solution above and had it write this data structure out to disk. When the solver program was given this "correct" file, it would then perform a root-mean-square error calculation,

$$e_{RMS} = \sqrt{\left(\frac{\sum_{i=0}^{n}\sum_{j=0}^{n}(u_{i,j} - c_{i,j})^2}{n^2}\right)}$$

where n is the length of one side of the grid, $u_{i,j}$ is the value of the calculated result at cell (i, j), and $c_{i,j}$ is the value of the analytical result at cell (i, j).

However, this does not help define a way for the algorithm to determine when to halt. We cannot use an analytic solution to determine if the simulation is close enough to a solution for every configuration we could solve for. First, there would be no point to doing this as it would imply that we already had the solution, and we will not always have the solution to an arbitrary problem. Second, comparing the computed solution to a verified solution on each iteration would be inefficient. Because of this, another method for determining when the simulation has converged is required.

The method I chose was to have the simulation keep track of a sum of squares while calculating an iteration. For each cell that it updates, it determines how much the cell's value changes. The simulation then sums the squares of all of these values and, at the end of each iteration, it divides this sum by the number of cells in the grid and compares that to a configurable threshold. If the scaled sum is lower than the threshold, it stops the simulation and returns. This allows the user to define a quality for the simulation by changing the threshold value.

2.3 Potential From Electric Dipole

Another correctness check is a simulation of an electric dipole, since the electric potential from an electric dipole is well understood. If two equal and opposite charges are separated on an axis by distance d, then at a point p far away the potential is

$$V = C\left(\frac{1}{r_0} - \frac{1}{r_1}\right),$$

where r_0 and r_1 are the distances from each charge, and C is a constant. With some rearranging,

$$V = C\left(\frac{r_0 - r_1}{r_0 r_1}\right) = \frac{Cd\cos(\theta)}{r^2},$$

where r is the distance from the center of charge to the dipole, and θ is the angle that the line from the center of charge forms with the axis [6]. The potential from an electric dipole thus falls off as $1/r^2$, which should be reflected in the solution given by the program if it is given a configuration describing a dipole.

If the solving program is correct, then if two charges are placed near each other, the numerically calculated potential should display this $1/r^2$ behavior. I wrote such a configuration and had the frontend output the potential at points increasing in distance from the center of the dipole, allowing me to fit the data to a $1/r^2$ curve using gnuplot.

2.4 Optimization and Multi-threading

The purpose of this program is to not only be correct, but to be correct quickly. In order to achieve this, I have tried three things in combination:

1. General program optimization, targeted towards 64-bit x86 machines
2. The use of the x86 SIMD instruction set
3. The use of multi-threading to split up the work among CPU cores

The first of these is pretty basic. The heavy lifting of the solver is written in C so that it can run directly on the processor. When the program is compiled, I instruct the compiler to optimize the program to the best of its ability, and the program is written in such a way that it can be optimized well and does not waste time doing unnecessary work.

As an example of this point, consider a 2-dimensional grid (2-D array), which must be iterated over (much like this program does). There are two ways to do this: either iterate row-by-row, or column-by-column. The end result is the same, and one might think the choice is arbitrary. However, when I tested the two different approaches, I found the row-by-row approach to be around 5 times faster. This is because of caching effects and the way that the CPU fetches data from RAM[3]. For a more detailed description of this, see Appendix B.

Another example involves data structure organization. If a program needs to store an array of objects, and each object has data associated with it (for example, current value, previous value, error, initial state, etc), one might be tempted to write something like the structure in Listing 1. The problem here is that if each object's `cur_val` are all calculated together (as is often the case, and is definitely the case for this solving program), then the location of the values that need to be loaded in succession by the processor are far apart. The performance hit from this, again, comes down to caching effects which are further explained in Appendix B.

```c
struct {
  double cur_val, prev_val;
  float err;
  int initial_state;
} objects[N];
```

Listing 1: Array-of-structures organization. Each attribute of a cell is declared as a field in the structure, and an array of such structures is defined. This can be an inefficient way of organizing data.

Fortunately, this also has an easy-to-make change that drastically improves performance: a change to structure-of-arrays organization, thus reorganizing the memory layout so that related and commonly-accessed-together values appear adjacent in memory[2]. This is shown in Listing 2. While it may be a less intuitive way to describe and program a simulation, the result is significantly faster code the majority of the time.

```c
struct {
  double cur_val[N];
  double prev_val[N];
  float err[N];
  int initial_state[N];
} objects;
```

Listing 2: Structure-of-arrays organization. Instead, now each attribute is its own array, thus placing them all in a contiguous region of memory. This organization can result in better performance.

2.5 SIMD Instructions

I have also made use of the SIMD (Single Instruction Multiple Data) instruction set. These instructions are available on modern 64-bit x86 processors[4][3]. Their purpose is to enable what's known as vector processing of data on x86 machines. This provides the ability to apply a mathematical operation to one or more arrays, and have that operation applied to each element. For example, I can have a 128-bit register filled with 4 values, $x = (1, 2, 3, 4)$, and another register like it, $y = (10, 10, 11, 11)$. I can then add them together to get $(11, 12, 14, 15)$, and this addition is issued with a single instruction. Furthermore, this takes the same amount of time as a single addition, because the processor can do the 4 separate addition operations in parallel[4].

I have written in the use of SIMD instructions in the solver program in order to both reduce branches and to parallelize the calculations required when calculating the next iteration of a given cell. I provided the ability to disable this functionality and instead do a non-SIMD style calculation in order to get benchmarks. A requirement of the SIMD code was that it give exactly the same result as the non-SIMD version (otherwise the implementation of the math would be incorrect). For this reason, when talking about the accuracy of a simulation, it does not matter if it used SIMD or not; the use of SIMD only affects performance.

[3]More instructions are added in most new releases of CPUs. There are different SIMD instruction sets available on different processor models. I have used the sse through avx instruction sets in this code.

[4]This is not the same as multi-threading. The processor does these 4 separate addition operations in parallel using 4 separate addition circuits, thus doing these operations in parallel within one core.

When fully optimizing code for a known environment, `gcc` and other C compilers may choose to output SIMD instructions in order to do computation even if the user does not do so explicitly. For this reason, in places where I refer to the mode of the simulation as being “non-SIMD”, I mean is that it does not use my manual SIMD instructions but it may still use compiler-generated SIMD instructions.

2.6 Multi-threading

Finally, I have added multi-threading support to the solver. This is done by splitting up the grid into subregions which are contiguous in memory, and spawning N threads via the `pthreads` threading library available on UNIX systems. Each thread is then given a region to work on, and they coordinate through shared state in order to decide when the solution has been reached. The initial implementation showed extremely poor results with multi-threading (often being drastically slower than the single-threaded version) due to the use of atomic shared variables. It was then re-written such that the code was unsafe but much faster. This is because each thread modifies a shared non-atomic variable that keeps track of the current total change per iteration after that thread itself completes a number of iterations over its region. This kind of shared memory access is much faster than coordinating the threads, but it is also technically undefined behavior in C. Fortunately, it is actually safe on x86; however there is potential information loss because a thread could overwrite another thread if they are trying to update the value at the same time. A more detailed discussion of this approach and why it is unsafe can be found in Appendix B.

2.7 Benchmarking Performance

Benchmarks were done by placing two calls to `gettimeofday(3)` around the core of the solver. This function has a granularity of 1 microsecond on Linux[7]. The benchmarking code returns both the number of iterations done and the difference between the two `gettimeofday` calls, which are then used to calculate the iterations per second. In order to take into account error from unpredictable scheduling effect from the operating system, each benchmark was run 300 times, and the standard deviation and mean were calculated, along with 95% confidence intervals. Several different computers were used: computer A was an Intel Core i5-3570K with 4 cores running at 3.4GHz with 16GB of RAM. Computer B was an Intel Xeon E5620 with 4 cores with 2 threads each, running at 2.4GHz, with 24GB of RAM. Computer B did not support all of the SIMD instructions that I used, so it was only able to benchmark the non-SIMD version of the code.

3 Results

All results describing the correctness of the simulation or discussing the closeness of the simulation to an analytical result are done using the single-threaded mode. This is because this makes the result of the simulation deterministic, and so it does not matter which machine the test was run on. For results that measure performance, the machine and options for the simulation used will be specified. There are times when the number of threads affected the correctness of the result.

Additionally, all results presented use the successive over-relaxation method for calculating the solution. This is because it converges more quickly on a solution that Jacobi iteration. Both methods produce the same results (as long as the Jacobi iteration runs for long enough), so the correctness results are unaffected by the choice of algorithm. Secondly, the performance measurements measure iterations per second, which is also largely independent of the choice of algorithm. Since successive over-relaxation is so much more effective at quickly converging upon a solution, it is presented as the program's primary mode of operation, with Jacobi iteration provided as a secondary and deprecated option.

3.1 The $\sin(x)$ Potential

Figure 2 shows the output of the solver program in single-threaded mode when given the $\sin(x)$ along one side potential, plotted with the Python library matplotlib. The

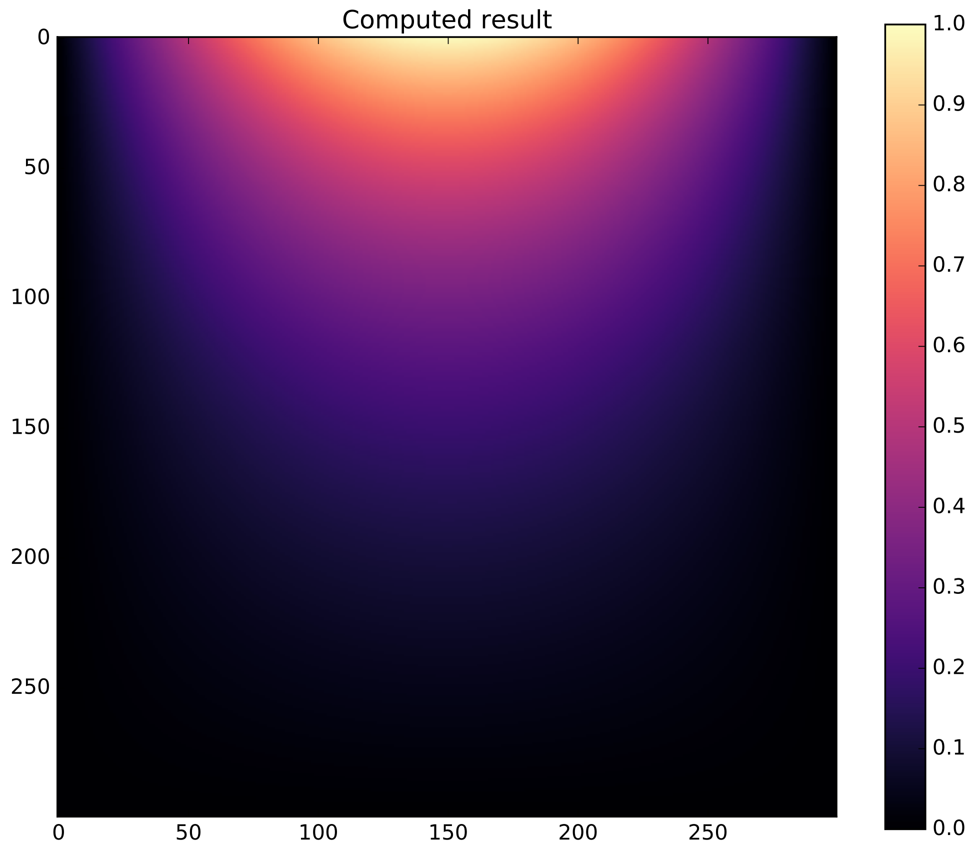


Figure 2: Result of simulation for the $\sin(x)$ potential, computed on a 300 by 300 grid.

configuration has a grid size of 300 by 300. It matches well with the analytical result, which is

$$\frac{\sinh(y\pi/n)}{\sinh(\pi)}\sin(x\pi/n),$$

and is shown in Fig. 3. The difference between the calculated result and the analytical result is shown in Fig. 4. The values shown in the difference map are small

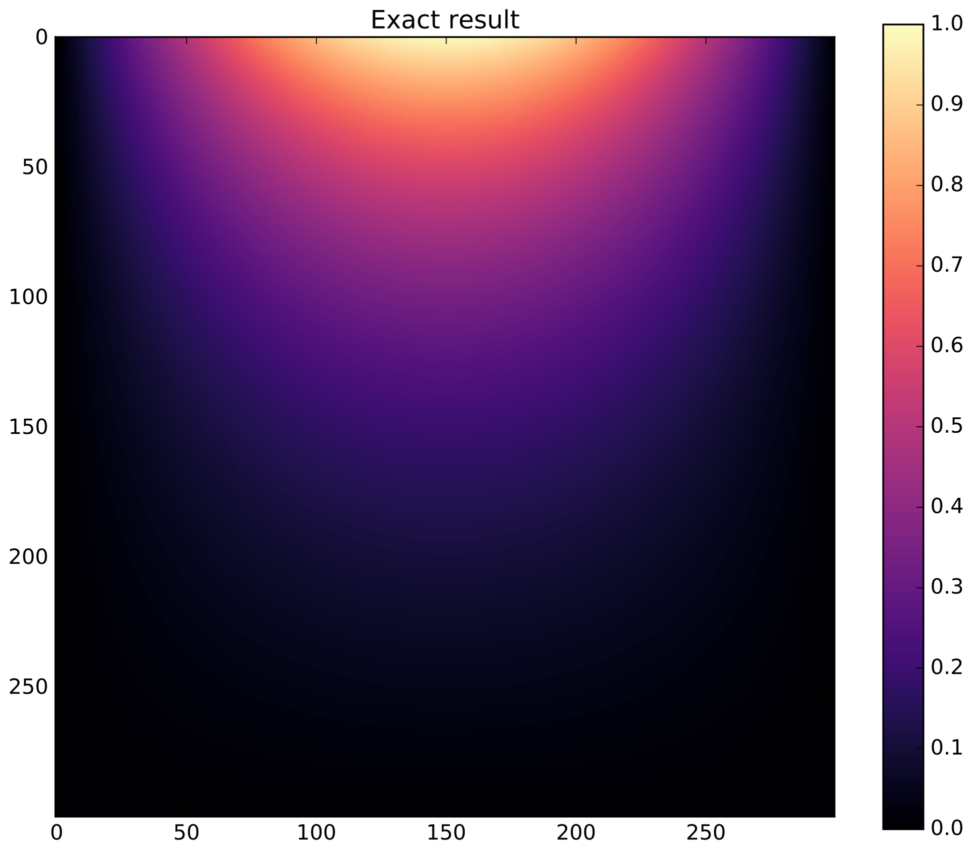


Figure 3: Analytical result for the $\sin(x)$ potential, with the same initial conditions and boundary conditions as the simulation.

compared to the values of the result in all locations except the corners at the top. The simulation does diverge somewhat in those corners, but never reaches values which are significantly different compared to the error from the analytical result. Using the root-mean-square error statistic described earlier, we have calculated the RMS error in this simulation to be 0.0025. The mean of all the cell values in this simulation is 0.19, so the mean error per cell is 1.3%. The error is small compared to the mean value of all of the cells, showing that the simulation result is of high quality.

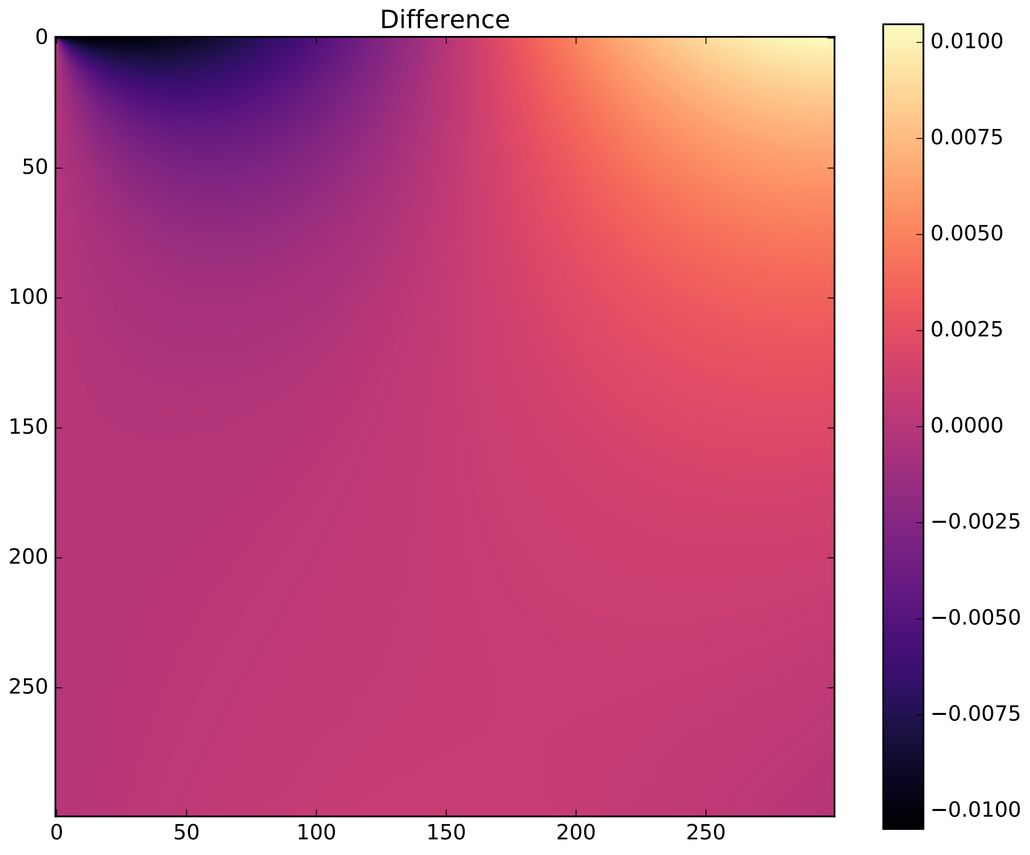


Figure 4: Difference between the calculated and analytical results for the $\sin(x)$ potential.

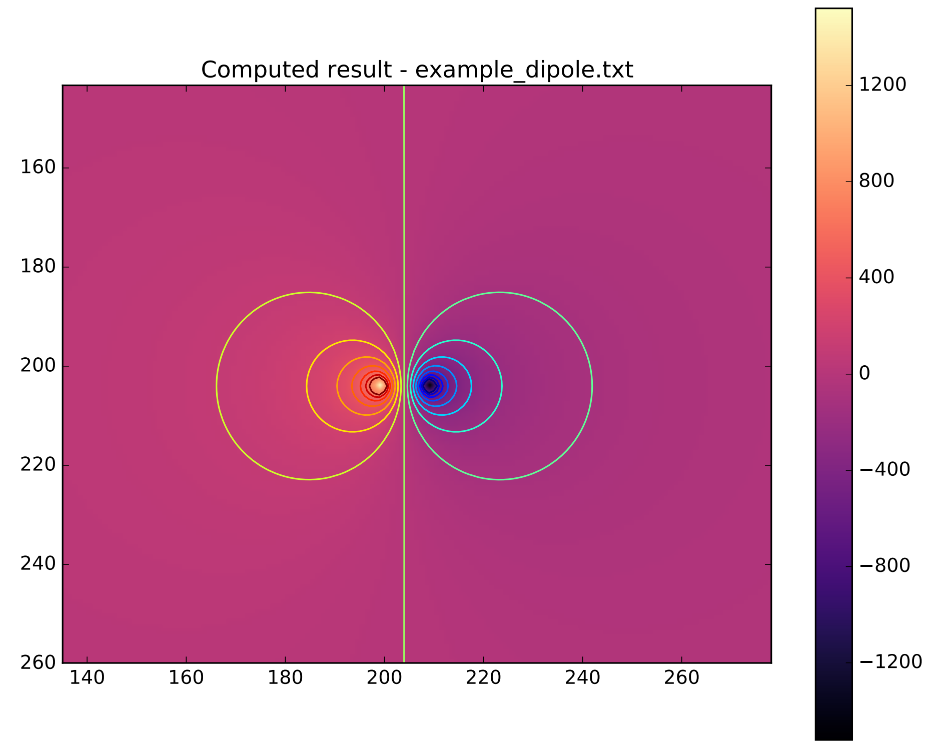


Figure 5: Result of the simulation for the dipole configuration, including contour lines. The vertical line in the center shows that there is a line of constant, zero-valued potential at points equidistant from the two charges, which is what one would expect to see from a dipole. Note that this view is zoomed in, as otherwise the features would be too small.

3.2 Fitting to a $1/r^2$ Dipole Potential

The dipole configuration had a grid size of 400 by 400, with all walls given a Dirichlet boundary condition of zero. Two charges were placed near the center 10 cells apart with an arbitrary but equal and opposite charge. The result of this simulation is shown in Fig. 5, with the addition of constant-potential contour lines. The simulation shows two opposite charges which near perfectly mirror each other. The vertical line in the center represents a equipotential contour with a value of zero, which is expected from an electric dipole. This is another excellent verification that the simulation is producing correct results.

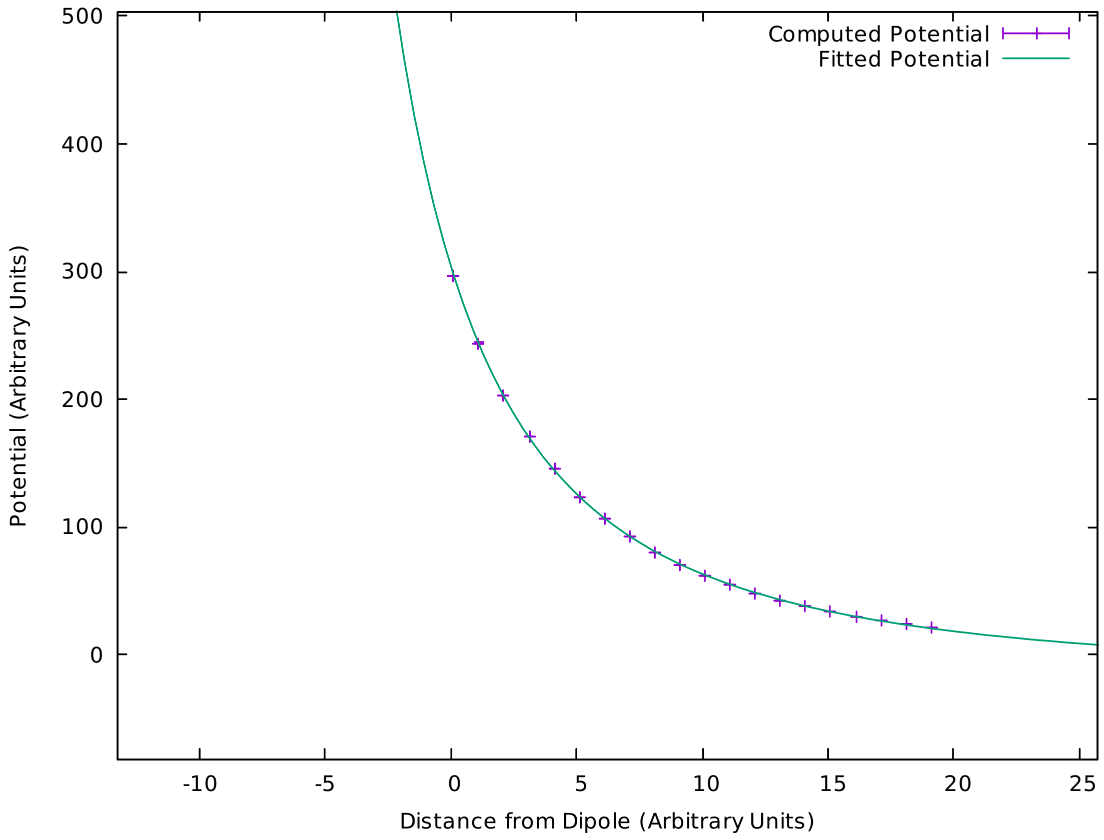


Figure 6: Fitting a $1/r^2$ curve to the calculated dipole potential. Both the units of the distance and the value of the potential are taken into account in the fitting by providing the necessary constants for the fit. The error bars are from the RMS error per cell as described earlier.

I then measured the potential at locations increasing in distance from the dipole in an arbitrary direction. These data were then plotted using gnuplot, and a line of the form $y(x) = A + B/(r - r_0)^2$ was fit to the data, which is shown in Fig. 6. The fit is excellent, confirming that the program had properly simulated the electric dipole potential.

The dipole is an example which lends itself to presenting another feature of the solver program. If passed the `-V` option, it will take the negative gradient of the result and plot that as a vector field. This has the effect of plotting the electric field of the result of the simulation, which for the dipole is shown in Fig. 7.

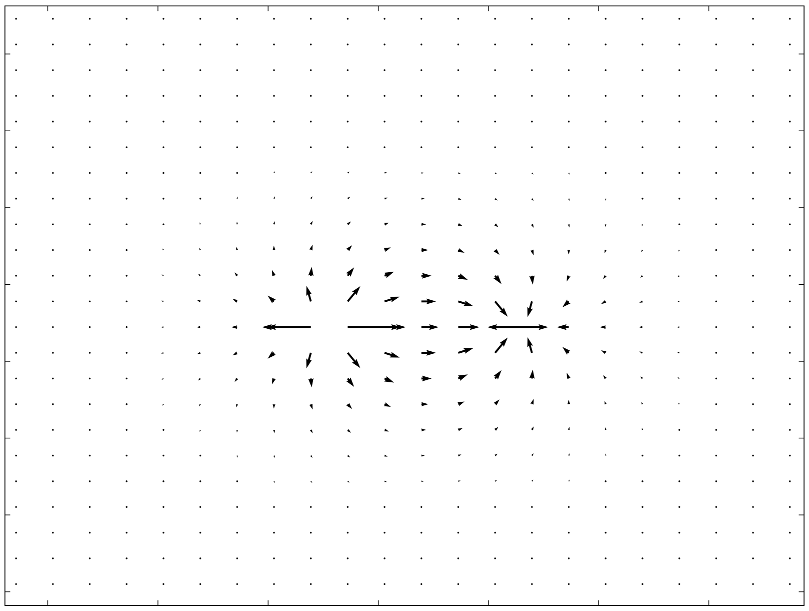


Figure 7: Result of program outputting the electric field for the dipole configuration. At the point of the negative charge, the field lines appear to point outward. They are actually pointing inward but are long enough that they cross each other to form this illusion.

3.3 Jacobi Iteration Versus Successive Over-Relaxation

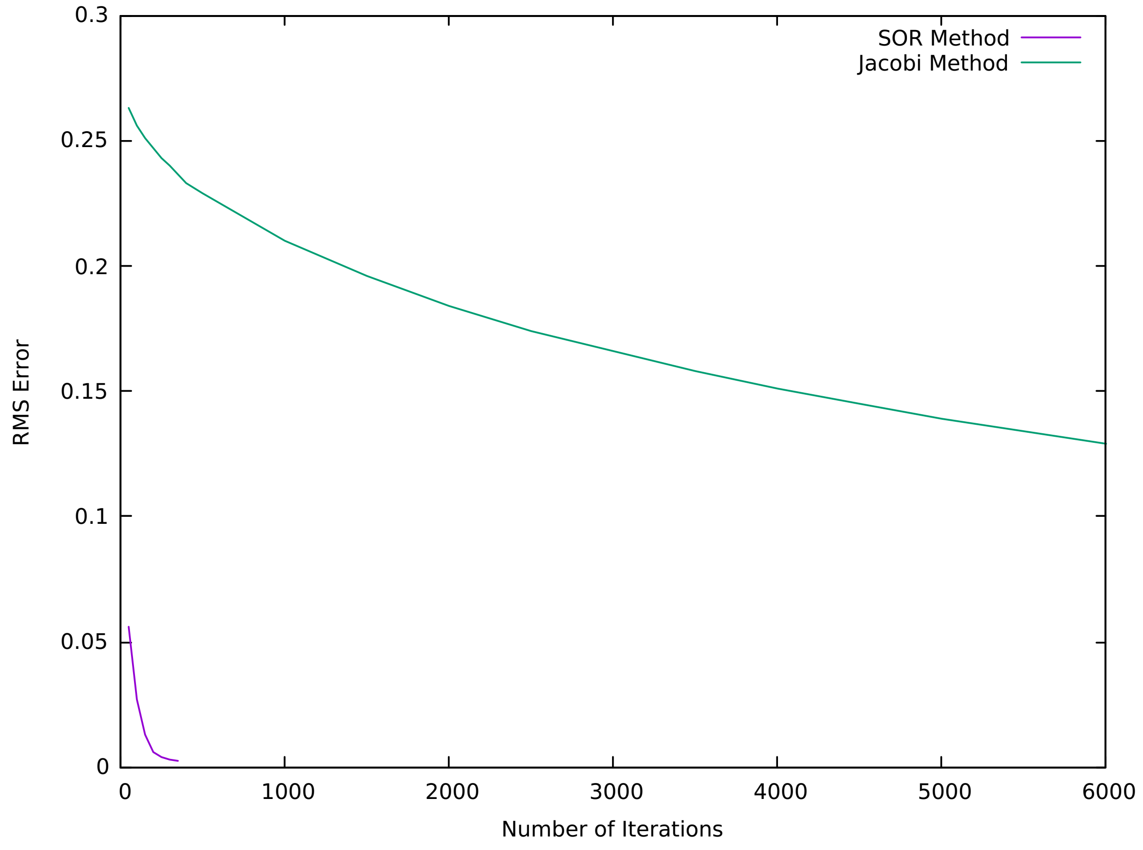


Figure 8: Convergence of Jacobi iteration and SOR. The SOR line is truncated once it reaches convergence. The measurements were made in increments of 50 iterations starting at 50 until 500, at which point the steps were done in increments of 500. The graphs do not start at zero because the error would be very large and would not assist in illustrating the difference between the two methods.

Figure 8 shows the convergence rates for Jacobi iteration and successive over-relaxation. The SOR method reached convergence rapidly compared to Jacobi iteration – around iteration 350, whereas Jacobi iteration at the same iteration was 100 times worse than SOR. This is the reason why I use SOR in most of my analysis; it reaches convergence so much faster that even if Jacobi iteration were capable of performing twice as many iterations per second, successive over-relaxation would *still* be much faster to find the solution. This is the power of the asymptotic behavior analysis of different algorithms.

3.4 Performance

Figure 9 shows the performance in iterations per second for a varying grid size for a given configuration, in this case the $\sin(x)$ along a wall potential. The performance tests were run using square grids of size 50 by 50, 300 by 300, and 1000 by 1000. For each grid size, 4 different simulation options were given: single-threaded non-SIMD, single-threaded SIMD, multi-threaded with 4 threads non-SIMD, and multi-threaded with 4 threads SIMD.

In the case of the 50 by 50 grid, the performance was similar for all cases except for multi-threaded with SIMD. I believe the reason that the SIMD multi-threaded version was slower was due to the manual SIMD code being less cache-friendly, resulting in the threaded versions having more contention. Both the 300 by 300 case and 1000 by 1000 case display similar performance behavior: the SIMD case is faster than the non-SIMD case when single-threaded, and the SIMD case is slower than the non-SIMD case when multi-threaded. This can be explained in the same way as it was for the 50 by 50 case. Another characteristic is that the multi-threaded case

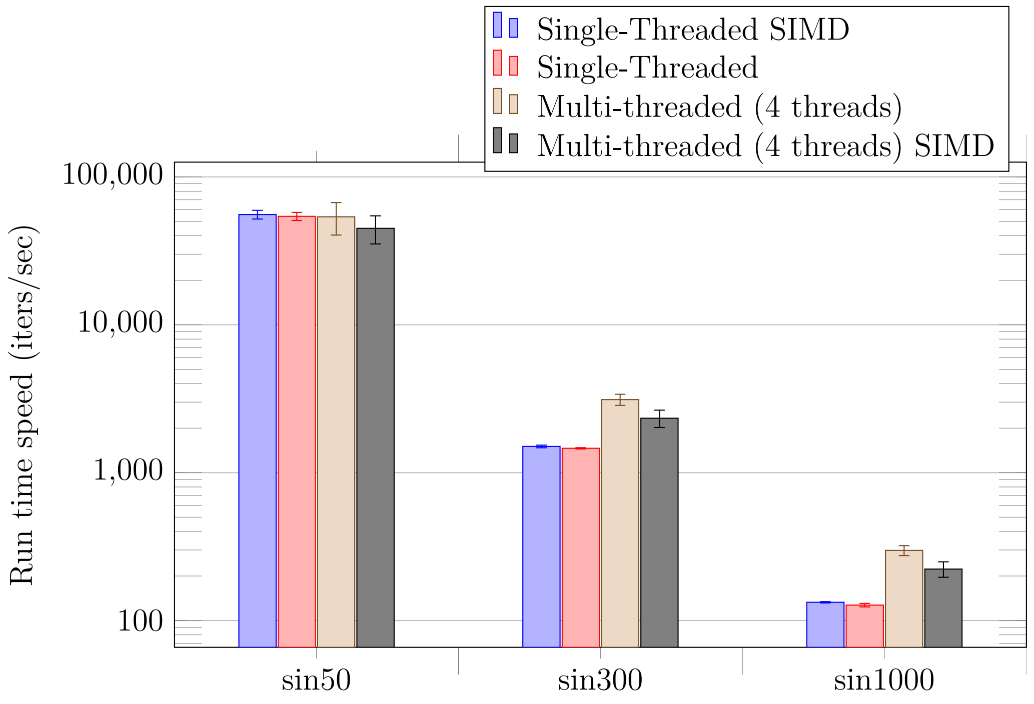


Figure 9: Different grid sizes for the $\sin(x)$ configuration, and their performance characteristics with different options for the solver. The performance also decreases rapidly as the grid size increases.

(with 4 threads) is faster than the single-threaded case by about a factor of 2. In fact, it is significantly faster than I expected it to be.

Additionally, the performance depends heavily on the grid size. This makes sense, as the amount of work needed to be done per iteration is proportional to the number of cells in the grid, and the number of cells in the grid is n^2. Another note is that the variance on the performance is significantly higher for all of the multi-threaded cases. This is because having multiple threads increases the program's dependence on the scheduling quirks of the operating system. Finally, the variance for the 50 by 50 case multi-threaded versions are significantly higher than all other variances for the other multi-threaded cases. I believe that this is because the overhead of managing threads compared to how much work they actually have to do is large in this case compared to the other cases.

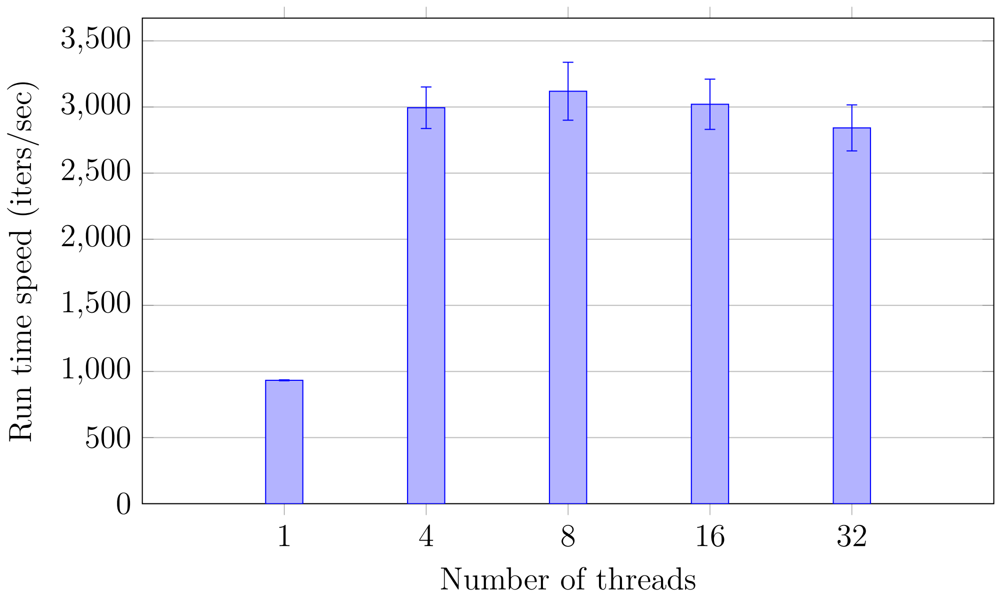


Figure 10: Performance results for a varying number of threads in non-SIMD mode. These tests were done on computer B, using the $\sin(x)$ potential with a grid size of 300 by 300. The performance increases until it reaches a maximum at 8 threads and then degrades after that.

Figure 10 shows the performance in iterations per second of the solver when run with the $\sin(x)$ potential with a grid size of 300 by 300 with a varying number of threads. This was done on computer B, and so was done only in non-SIMD mode. The results show a significant improvement in performance when using multiple threads compared to the single-threaded case. One may naïvely have expected the performance to scale linearly with the number of threads; however, this would be unlikely. As the number of threads goes up, the amount of work that can be done in parallel would seem to scale with the number of threads, until one considers processor cache effects (specifically cache coherency and cache size limitations). This is seen in the case of 4 threads, as it is not 4 times as fast as the single-threaded case, only about 3.2 times as fast. Increasing the number of threads quickly starts giving diminishing returns, as the 8 threads case is now only slightly better than the 4 threads case. Recall that computer B, on which these tests are run, has 4 cores with 2 threads each, totaling 8 possible threads running in parallel. After the 8 threads case, we start to see a drop in performance; the 16 and 32 threads case are worse than the 8 threads case. This is because the machine cannot run more than 8 threads in parallel, so in order to have 16 threads it must rely on the scheduling of the operating system in order to get work done. In the case of many threads, each doing heavy computation and memory accesses (as is the case here), it does not usually help to increase the number of threads beyond what the machine can handle, as is shown here.

Furthermore, Fig. 11 shows that it is disadvantageous to run the simulation with a large number of threads. The RMS error of the simulation compared to the analytical result increases rapidly with the number of threads. This is also reflected visually in the plotted result of the simulation, which is noticeably wrong in the case of 16 or higher threads. I believe this is the result of the way the threads are synchronized, or more accurately, how they are not. The threads run mostly in isolation, sharing the grid and doing operations on it without waiting for other threads to complete any of their work. This means that if one thread runs faster than the others, it may complete a different number of iterations than the other threads in a given amount of time. This would result in some sections of the grid receiving more iterations than others, which could have the effect of invalidating the simulation. The alternative method would be to synchronize the threads, and have them wait for all threads to finish an iteration before moving on to the next one. Indeed, this was the original design of the system, and I found it to be so slow compared to single-threaded that I changed the design around to the less accurate but much faster design that is shown here.

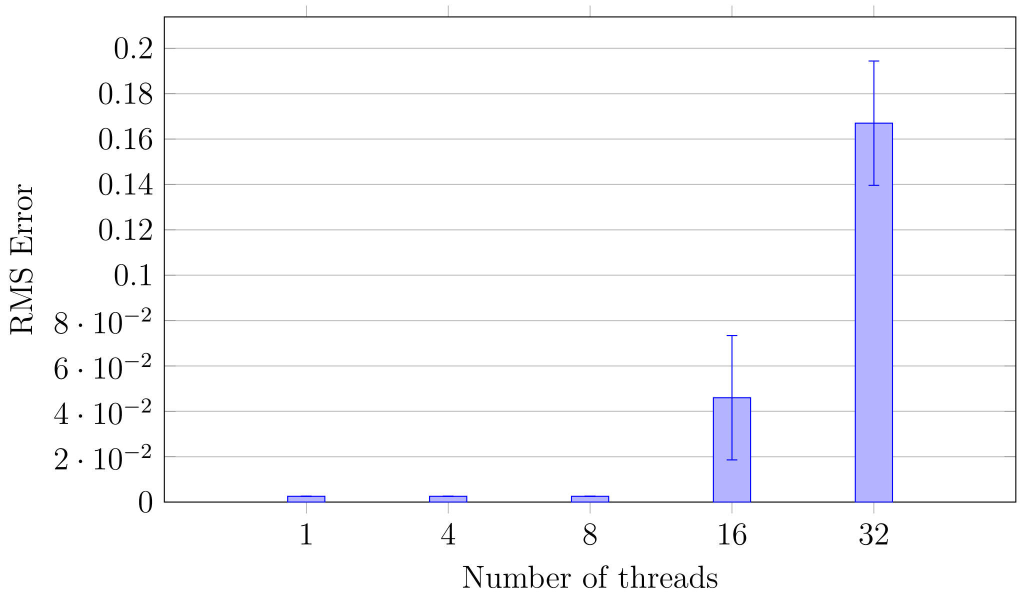


Figure 11: Error results for a varying number of threads in non-SIMD mode. These tests were done on computer B, using the $\sin(x)$ potential with a grid size of 300 by 300. The error appears to be roughly proportional to the number of threads once the number of cores on the processor are saturated.

4 Conclusion

This program correctly solves certain boundary value problems, enabling simulation of complex electrostatics situations. It does this in an efficient manner by leveraging proper programming languages and techniques, and by utilizing a basic understanding of modern processor design.

4.1 Correctness

The most important aspect of this program is that is produces an accurate result, no matter how fast it runs. I was able to use a problem for which I had an analytical solution to confirm that the simulation was producing accurate results and found that the results were of high quality for the single-threaded and low number of threads cases. Additionally, I was able to use the solver program to accurately calculate that the electric potential from an electric dipole drops off as $1/r^2$, further confirming the correctness of the program.

4.2 Performance

After verifying that the simulation is correct, I was able to then turn my attention towards improving the performance without reducing the accuracy of the simulation. Most of this was easy to do without reducing accuracy, as it was simply transforming one valid implementation of the code into another and measuring to ensure that my new implementation did in fact cause an improvement. Most of this was done by improving cache locality and reducing memory accesses. I have left out a lot of this optimization process for the sake of brevity; however an overview of it and the concepts used can be found in Appendix B and Appendix C.

The next significant performance gain attempt that I had made was the use of SIMD instructions. I was hoping for a much more significant performance boost than I saw, which was rather minuscule in reality. The rather small improvement was not surprising in the single-threaded case because I knew that `gcc` utilizes SIMD when it can, but I did not expect to see the manual SIMD instructions degrade the performance in the multi-threaded case. I speculated that this is caused by my manual SIMD code being less cache friendly, which is tolerable in the single-threaded case and allowed a small performance improvement, but in the multi-threaded case the threads are all contending for limited cache space, so the additional memory usage slowed them down.

The multi-threading actually improved the performance significantly more than I expected it to. I had predicted that the algorithm would be cache-bound due to the large amount of memory accesses that it makes. This often results in insignificant performance improvements (or downright performance degradation) when adding multiple threads to a single-threaded program. Of course, the cardinal rule of any scientific experiment (including attempting to optimize a program) is to measure and compare results. Clearly the multi-threaded case is significantly better in performance, with a trade-off in accuracy, which may certainly be acceptable.

4.3 Future Work

There is still a significant amount of optimization work that can be done on this program. A careful reading of Intel optimization manuals would likely provide numerous ideas for how to further improve the code of the program. The manual SIMD code could also be improved, either by utilizing more up-to-date instruction sets or by improving the existing algorithm. The multi-threaded code could also be improved, by allowing the threads to communicate in a limited way in order to better synchronize their work to improve the quality of the result, hopefully without degrading the performance too drastically. An additional approach could be to use the multi-threaded case for a few hundred iterations before switching over to the single-threaded version in order to smooth out and cleanup the results. This would have a significant advantage for large simulations, as the multi-threaded code could get the result most of the way before letting the single-threaded code produce an accurate result in a shorter amount of time than if the single-threaded code ran from the start.

Another significant way in which to improve the real world performance of the program, be in addition to optimizing the speed at which the program can calculate an iteration, could be to reduce the number of iterations required to converge on a solution. Indeed, this is what the method of successive over-relaxation achieves, but there may be ways to further improve this. One such method could be to improve the initial “guess” of zero everywhere by dividing the initial grid into a much coarser grid and running a few iterations on that before returning to the full grid and completing the simulation as before.

Appendices

A Program Usage

Compiling the engine requires `make` and a modern `gcc` (or `clang`, although `clang` is untested), and can be done by simply invoking `make`. This will also create an executable file, `solve`. This is a script that invokes the Python example frontend so that it may be used.

For more details on using the Python frontend or the C backend engine, see the man pages (reproduced on the following pages). There are several example files, including `example_sin.txt` and `example_dipole.txt`. For more details on writing configurations for a desired simulation or experiment, see the man pages.

The source code also contains a directory named `tests`, which contains Python code to generate verifier data files for the `example_sin` configuration. Finally, all of the data written on in this document are present in the source directory, under `results`. The source code for the C library is in `engine`, and the source code for the python example frontend is under `frontend`. The source code and git revision history is available at `http://github.com/dbittman/statics-solver`.

NAME

solve - Solve Discrete Poisson Equation

SYNOPSIS

`solve [-V] [-m` *`method`*`] [-v` *`verification-data`*`]` *`configuration-file`*

DESCRIPTION

Solve Poisson Equation given boundary conditions on a discrete grid. Capable of solving physics problems that reduce to a boundary value problem. Operates on the provided configuration file, producing a 2-D array describing the calculated potential.

-V

Produce a vector plot, where the vectors are the negative gradient of the potential.

-m *method*

Use method *method* when solving. Current supported values are `jacobi` for Jacobi Iteration, and `sor` for Successive Over-Relaxation.

-v *verification-data*

Use the data file *verification-data* to compare with the generated potential. The format for this file must be that of the Python library pickle serializing a 2-D array.

CONFIGURATION

The configuration file format is specified by a series of commands on lines. A single line can contain at most one command. A command must be contained within one line. Comments begin with a #, and comment out the rest of that line. Valid commands are as follows, where italics indicates something to be replaced by one token:

`gridsize` *`size`*

Set the size of the grid. The grid is always a square, and this command **must** come before any other.

`cell` *`coords`* `initial` *`value`*

Specify initial value of a cell inside the grid. Analogous to a point charge.

`dirichlet` *`coords coords`* `=` *`value`*

Specify a dirichlet boundary condition along the interpolated straight line from the first set of coordinates to the second with value *value*.

`neumann` *`coords coords direction = value`*
: Specify a neumann boundary condition along the interpolated straight line from the first set of coordinates to the second with value *value* across the boundary of the cells specified by *direction*, which may be one of `left, up, right, down`.

The values of *`coords, size, direction`* must all be one token, that is, they may not contain any whitespace. The contents of *`value`* and *`coords`* are as follows:

`value` is an `expression`.

`coords` is an `expression` followed by a comma, followed by an `expression`.

An `expression` is a valid Python expression, with access to the python math library, the current grid class, and the current position in the grid as specified by `x` and `y`. For example, `math.sin(x*math.pi/grid.len)` is a valid `expression`.

AUTHOR

Written by Daniel Bittman (`danielbittman1@gmail.com`). Please submit any bug reports to this email address.

COPYRIGHT

NAME

solver - Library to solve Discrete Poisson Equation

SYNOPSIS

```
solver.h
    SOLVE_METHOD_JACOBI, SOLVE_METHOD_SOR
double solve(struct grid *grid, int method);
void init_grid(struct grid *grid);
struct grid {
    int len;
    int iters;
    float **values;
    float **value_prevs;
    float **initials;
    uint8_t **dirichlet_presents;
    float **dirichlets;
    uint8_t **neumann_presents;
    float **neumanns[4];
};
```

DESCRIPTION

This library provides fast solving of a 2-D discrete boundary value problem using the Poisson equation. The full process is to define a `struct grid`, and set its `len` field. Then call `init_grid` on the grid to initialize the 2-D contiguous arrays. After that, call `solve` and pass it the grid and a method, either `SOLVE_METHOD_JACOBI` or `SOLVE_METHOD_SOR`.

The `solve` function will return when complete, returning back a value describing how confident it is in its result (values closer to zero are better). The `iters` field in the grid will have been updated to indicate how many iterations the program took. The `values` field points to a 2-D array of size `len` by `len` containing the solution.

AUTHOR

Written by Daniel Bittman (`danielbittman1@gmail.com`). Please submit any bug reports to this email address.

COPYRIGHT

B Computer Program Optimization

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

Donald Knuth

When optimizing a program, there is an important cardinal rule to follow which at times is easy to forget in the face of increasingly interesting and complex minor optimizations designed to squeeze every bit of performance out of your computer. Unfortunately, due to the complexity of modern computer hardware, it is difficult to predict if a given micro-optimization will actually be beneficial or not. For this reason, the most important rule when taking a working program and making it fast is *benchmark everything*[5].

The field of program optimization is huge. There is a small number of people who, for a given platform, are really *good* at optimization. There is an even smaller number who are truly experts. I do not fit into either of these categories. For this reason, the program that I have written here is *decently* optimized. I believe that the maximum throughput is not orders of magnitude better than what I have achieved, however, there is still a lot of work that can be done.

[5]besides *don't break it*, of course. An incorrect program is worthless, no matter how fast it is.

B.1 Lazy Programs Are Better

Before taking up a sharp cutting tool and carving away a program to make it run faster, it helps to examine what exactly is the program trying to accomplish and how. There are often multiple ways to solve a problem, and some of them may be more efficient than others. In my case, I utilized two different methods of solving a problem: Jacobi iteration and successive over-relaxation. One of them was significantly faster than the other not because each iteration of the program was quicker but because it had to do less iterations. It was using a smarter algorithm.

Computer scientists talk about algorithmic complexity using order notation. For example, given a random list of numbers, if I want to test to see if a certain number if in the list, I need to look through the whole list in order to check if it is there. This is an $O(n)$ operation, meaning that if the list is of size n, I have to do n operations to determine if the item is there or not. Data structures commonly have operations associated with them, and those operations (such as `find`, `insert`, etc) have associated time costs. Choosing the correct data structure for a problem is one of the first steps towards correctly organizing a program for it to be efficient. If a program is constantly searching through a list to find values when instead a simple hash table could be used, the program will run slowly. A program should not do additional work when it does not need to. Only after data structures have been properly designed and algorithms properly worked out should *optimization* be done.

B.2 Freebies

There are some optimizations which can be used in high confidence and are also easy to do. The first, and most obvious, is to use a fast language. While writing in `C` may be a chore compared to writing in Python, the resulting code will be much faster. This is the reason behind the organization of this program: the frontend (which has no performance requirements) is written in Python because parsing the configuration file is much easier. The backend needs to be fast, so it is implemented in `C`.

A quick and rough experiment can easily show that this is true. Credit to my friend Chris Milke for doing this test. Write the following code in both python and in C: initialize a variable to zero. Then loop from zero to some large number (we used 123,456,789), and add that iteration to the variable. After the loop, print the variable. The resultant python program took 14.25 seconds to run on my computer, but the C code completed *instantaneously*. Why is python slower than C in general? Because python is an interpreted language. The code that you write is read by another program and executed by that program. In contrast, C code must be compiled into machine code. This is then run directly on the processor, which cuts out the middle-man of the interpreter.

When using C, there are some more things that you can get for free: compiler optimizations. When compiling a C program, the command looks something like this: `gcc -Wall -Wextra foo.c`[6]. This will result in a compiled C program

[6] As an aside, one should **always** compile with -Wall and -Wextra, and try to eliminate all warnings from your program. Warnings are warnings for a reason, and should rarely be ignored.

(a binary file, containing the machine code), but it will be unoptimized. There are reasons why one may want an unoptimized binary (they are typically easier to debug, for example), but generally when you compile your program for actual usage you will want to optimize it. An aggressive optimization compilation command may look something like: `gcc -Wall -Wextra -O3 -ffast-math -mavx`, as a start. The main flag here is `-O3`, which enables almost every optimization that gcc can do.

Going back to the example from earlier, the optimized version of the C program completed instantly, but the unoptimized version took 0.33 seconds. The unoptimized version is still 43 times faster than the python code, but what was the optimizing compiler doing to make it so much faster with `-O3` on? Part of optimizing is understanding why something is faster, so we should look at the assembly code. On UNIX, this can be done with the `objdump` command. Table 1 shows the annotated assembly from the function `main`. If you do not know x86_64 assembly language, you will at least notice that the optimized version is much shorter. If you do know how to read the assembly, you will notice that the unoptimized version is doing almost literally what the C code says to do: set a variable to zero, and iterate from zero to a big number, adding that iteration to the variable, and finally printing the variable. The optimized code just prints a large number – which happens to be the result of the sum. The compiler had pre-computed the answer beforehand.

Even if this is an unfair comparison to make with the python interpreter in that the C compiler "cheated" by pre-computing, consider two points: firstly, the python intepreter could do the same thing, although it may have to do it every time the program is run. This is the beauty of compiled languages. The compiler can do the work once up front. Secondly, even the unoptimized code, which is doing the exact same logical steps as the python version, was vastly quicker.

Unoptimized	**Optimized**
`mov [rbp-0x10], 0x0`	`movabs rsi,0x1b131147ee6b52`
`jmp .loop_end`	`mov edi,0x400594`
`.loop_top:`	`xor eax,eax`
`mov rax, [rbp-0x10]`	`jmp 4003c0 <printf@plt>`
`add [rbp-0x8], rax`	
`add [rbp-0x10], 0x1`	
`.loop_end:`	
`cmp [rbp-0x10],0x75bcd14`	
`jle loop_top`	
`mov rsi,[rbp-0x8]`	
`mov edi, 0x4005b4`	
`mov eax, 0`	
`call 4003c0 <printf@plt>`	

Table 1: Unoptimized and optimized assembly from the sum-a-lot-of-numbers example. The unoptimized version loops and does the work. The optimized version has been pre-computed.

This is just a simple example, but a good C compiler can drastically improve the performance of a program. Here, it made a program that took a third of a second finish instantly.

B.3 A Model of Processor Design

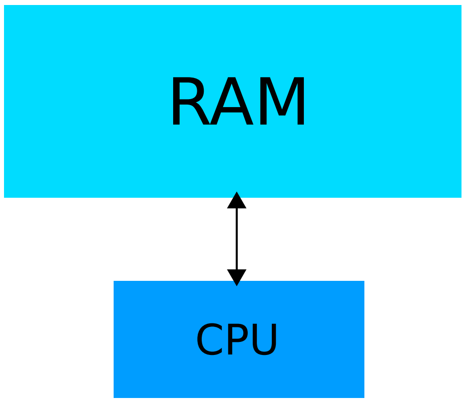
When first learning to program in a language that does more accurately display what is going on inside of a computer, it is helpful to have a mental model for what is going on during each simple operation that a programmer might want to do. Since programs are sets of instructions which define transformations on data, it is tempting to think about a model such as in Fig. 12a, where there is some nebulous device which follows your instructions and is connected to a large amount of memory in which your variables and data structures live.

Unfortunately, while this is a convenient abstraction to think about when programming up an application, it is not actually what is going on[7]. A more realistic model is shown in Fig. 12b. There are several key takeaways from this. One of them is that RAM is *really slow*. The second is that the processor chip is aware that RAM is really slow and so it aggressively caches everything that it ever gets out of RAM. This diagram shows the actual detailed level of caches that cores have on them, and that they have a shared cache, but most of that complexity is not needed for a basic understanding. When one makes first steps towards optimizing a program, it is vital to think in the more complex model, because a huge amount of time will be spent trying to optimize the program according to how the processor actually works.

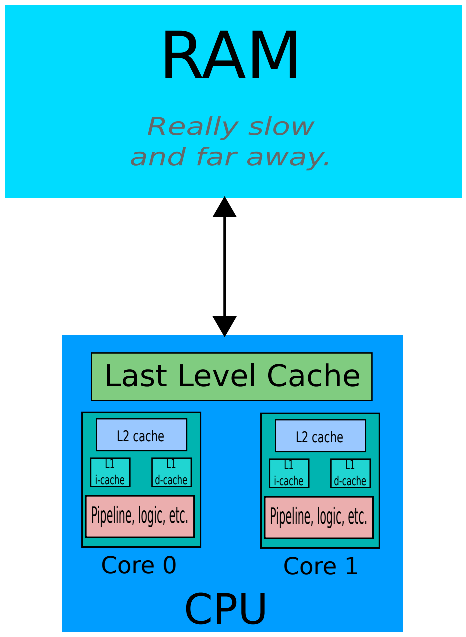
B.3.1 Caching

On modern processor chips, the vast majority of the silicon is used to implement a memory cache. This gives an indication of how important it is. Think of the cache as a smaller, but much much faster memory that the processor accesses whenever it

[7]I do not claim that what I am about to describe is what is "actually going on" either, but it is at least significantly less of a lie.



(a) Simplistic view of CPU and RAM.



(b) More realistic view of CPU and RAM.

Figure 12: View of programming that programmers like to have (a), versus the the more realistic and complicated view (b). The cache system is more complicated than will be described here, and the pipeline logic is also a significant factor in optimization which will also be glossed over.

would access RAM. If it finds the value in the cache, reading a value can take between 1 and 10's of nanoseconds. However, if it does not find the value in the cache, it must then perform a read from memory, which can take hundreds of nanoseconds. For scale, a single instruction often takes less than a nanosecond to do. This means that if a program makes a lot of memory accesses, the processor will spend a lot of time waiting and not much time calculating.

Earlier in this document, I described a phenomenon which describes how important caching is. Given a 2-D array, there are two ways to iterate over the whole thing: line-by-line or column-by-column. On my computer, the line-by-line method was five times faster than the alternative. This has to do with the cache, and specifically, *cache lines*. When reading a value out of memory for the first time, the processor actually reads many values, something like 64 bytes worth of data. This is a fact of the design of the hardware, and is partly because if you read a value x from RAM, the processor expects you will want to access memory that is nearby x in the near future. Thus, if you iterate over an array line-by-line, when you access the first element of a line you are actually reading in multiple elements of the array into the cache at once. When the processor then tries to access the second element of the line, it finds that it is already in the cache, making that access much faster than if it had not been in the cache. If you iterate over the array column-by-column, you read the first element of the line, which loads several elements from that line into the cache, and then *those values are ignored* and the first element of the second line is read. This means that when reading the first element of the lines, each time the processor has to fetch the value from RAM and does not get a chance to make use of the cache. By the time an entire column has been read, the cache may be full, forcing the processor to remove old values from it. Then, when the program reads the second element of the first line, that data are no longer in the cache. This is extremely slow.

If possible, try to keep the amount of memory that the program accesses small. Try to organize the access patters such that it is something that the processor is good at handling (like the iteration direction choice above). I have utilized these methods in my solver program. An example for keeping memory footprint small was to reuse variables inside the grid that I knew were safe to reuse. Additionally, I used `floats` instead of `doubles` wherever possible, as this reduces memory usage as well. Finally, I made sure to access data in a sequential manner and I ensured that the memory I was accessing was as contiguous as possible.

B.4 Structure of Arrays

While I had written code that utilized the x86 SIMD extensions, this was not particularly necessary. Modern compilers are capable of producing executables which use these instructions themselves, without the help of the programmer. Of course, a skilled programmer may be able to write a much better version than the compiler can because they have much more context for what the program is doing, but in general, the compiler can produce a somewhat reasonable output a lot of the time. However, one thing that it cannot do effectively is reorganize data access patterns into a more CPU friendly manner.

This is the reason why it is, at this point in compiler technology, worth it to restructure performance critical code to help out the compiler and the processor as much as possible. The structure-of-arrays versus array-of-structures design options are an example of this. As I described in Section 2.4, there can be a significant improvement from restructuring to a structure-of-arrays style of data organization. This is due to, for the most part, the SIMD nature of data processing. The code emitted by the compiler is likely using SIMD style data accesses, and they are helped tremendously by being able to load a lot of the same value from different cells at once. Because organizing the data in a SOA method places all of the values that have the same meaning for each cell next to each other, there is a benefit seen both in the way the code is trying to access the data and in the way the processor is caching it. Because the simulation uses neighboring cells (half of which are directly adjacent to the current value), organizing the data in a SOA manner means that it is likely that half of the neighboring cells will be in the same cache-line as the current cell, thus speeding up their memory accesses drastically.

B.5 Multi-threading

Multi-threading a program is a solution to performance problems that is commonly thrown out in a loose way. Multi-threading something is typically only effective when there are multiple largely independent pieces of work to be done in a system. It does not always improve performance. This is a situation, like any other performance question, where it is important to measure before and after and determine if it is worth it.

B.5.1 Undefined Behavior Land

One huge problem in multi-threaded programming is data synchronization. Consider a simple example, where there are four threads which are all sharing a variable named x which initially starts with a value of 4. If all threads execute the code `x++`, what are the possible values of x? Ideally, just 8, but that is not the case. The reason is because `x++` is really three separate operations: 1) read x into the CPU, 2) increment the value inside the CPU, and 3) write the new value to x in memory. So what happens if two threads read x at the same time? They both get back 4, and they both increment it internally to 5, and write back 5, resulting in two increment operations appearing as if only one had happened.

The answer is, x could be 5, 6, 7, or 8, and there is *no way to predict which one.* At least one increment will happen, and that is all that can be known[8]. This is an example of undefined behavior and is common in multi-threaded programs.

Fortunately, C11 provides a simple solution: declaring variables as *atomic*[9] (for example, `_Atomic int x`). If x in the example above were to be declared this way, then the answer to the above problem is simply "just 8." The advantage is that there is no loss of information when multiple threads operate on a single shared variable. The disadvantage is that this is slower because it effectively "locks" that part of the

[8]Technically, this is not really true, as this is undefined behavior which is quite literally undefined, and so your program could do anything from incrementing x to 9 to selling your liver on the black market (though this is admittedly unlikely).

[9]The name referring to small indivisible units, since *atomic operations* are indivisible (all or nothing).

memory and prevents other operations from occurring until one completes. This is supported in hardware for integer types, but not floating point types. This makes atomic floating point variables particularly slow because they need extra locking, and therefore extra memory accesses and CPU time for every access.

This was the problem I ran into when designing the multi-threading in the solver application. There was a shared floating point value to keep track of how much the simulation had changed after each iteration. This ended up being slower than the single-threaded case. By minimizing the shared state between threads and limiting the communication of floating point values between threads as much as possible, there was a huge performance improvement.

B.5.2 More Threads Means More Data

An additional concern with multiple threads is that you can run out of cache space more easily. Now, instead of sequentially operating on some block of data (because sequential data access is fast, so that is what you would do, of course), there are multiple threads working on different parts of memory at the same time. This means they are fighting over cache space. It is entirely possible for adding threads to make something *slower*, because of this contention. Again, the key comes down to measuring. It is hard to know ahead of time what will help and what will not. Multi-threading only makes things worse, since now the program can be run in a nondeterministic manner: sometimes some threads will be slower, sometimes they will be faster. This makes it even harder to predict, and more complicated to measure.

B.6 Branch Reduction

In programming, any statement in the language which will cause the flow of execution of the code to change based on a condition is called a branch. This includes if statements, while loops, etc. Many times these cannot be avoided, but there are also times when the code can be restructured in a way to reduce their number. This can be useful, because branches are often slow. This is partially because of caching (instructions for the CPU are cached as well as data, so if it suddenly jumps far away due to a branch, it has to load instructions from memory, which is slower). The real reason that branches are slow has to do with pipelining in the CPU, which I will not go into here. The summary is that reducing branches is good, and organizing the required branches in a simple way will probably be good enough for the processor to optimize them reasonably well.

B.7 Concluding Remarks

The most important idea when optimizing is to keep in mind that you may be wrong, so measure everything to ensure that the optimization is actually an optimization. Secondly, a highly optimized but slow algorithm will still be slow for large enough inputs. Fast processors do not beat order notation, and searching a hash table for a value ($O(1)$) is *extremely likely* to be faster than looking it up in an array ($O(n)$). Thirdly, make sure that there is no hidden cost in using a data structure, avoid repeated work that does not need to be repeated, and be aware of some basic reasonably fast algorithms (sorting, for example). Fourth, use a programming language which compiles to direct processor instructions (C, C++, among others), and use the optimization options for the language. Fifth, organize the program's data so that accessing it is typically sequential, and try to design the layout of data such that it is easy for the compiler and the processor to optimize. Sixth, multi-threading can help, but it can also be tricky to get right. Finally, work on reducing memory footprint, branches, code complexity, and complicated slow math if possible.

That said, programs should be written for correctness and clarity first, even if this means sacrificing some performance early on. However, there is a limit to this. Faster algorithms should be used, and we should try hard not miss Donald Knuth's 3%, as long as we remember to measure what we have done.

C Program Implementation

The procedure of the program implementation was completed in multiple phases designed to section off the different aspects of designing the program into more manageable pieces while allowing me to ensure that each transformation that I made to the program was one such that the program did not lose functionality or correctness at any stage. Before writing any code, I had a general understanding of the math behind the algorithm. This did not mean, however, that I was completely sure that the implementation would be correct, thus necessitating testing. As discussed previously, I designed several test cases for which I knew what the correct solution was.

The first iteration of the program was a simplified and not optimal version written entirely in C in order for me to confirm the correctness of my approach. This version made no attempts to be fast and had the boundary conditions and grid parameters hard-coded into it. I implemented Jacobi iteration and SOR, both of which arrived at the correct result for the $\sin(x)$ potential along a wall.

I next turned my attention to the question of how to interface with the program. Hard-coded boundary conditions are not desirable, and I quickly found it bothersome to have to modify the code in order to change the conditions to further test the program. I decided to use a python script to process a configuration file and then call into a C library which would do the heavy lifting. This is a common approach when dealing with processor intensive simulations because it allows the speed of C while allowing for the ease of python scripting in order to setup the simulation and process the results. This necessitated defining a structure in C and a class in python which both contained the same data in a layout that both C and python could agree upon. This ended up being not too complicated, except for the problem that python would not allow me to define the memory layouts for the actual grid. To solve this problem, I had the C library export another symbol which would initialize the data structures contained within the grid structure.

The python code was written to act as a parser for a configuration file and evaluate the result of calling the solver library. Parsing the configuration file is not complicated, so that code was all written without external libraries. Evaluating the returned data was done with a combination of the `numpy` and `matplotlib` python packages, allowing me to process and display the results.

Once I had defined and written this interface, I turned back to the C code to improve the performance. The biggest performance increase was seen by changing the data layouts to be a contiguous block of memory. Previously while making sure my understanding of the math was correct, I had simply defined the layout of the cells of the grid to be an array of arrays (since it is 2 dimensional), but not necessarily contiguous. After changing the allocation scheme, the performance improved dramatically. The second most significant improvement came from changing the C code from an array-of-structures style data layout to a structure-of-arrays layout, as I discussed earlier. This resulted in another significant performance boost.

After fixing up the general algorithms, I turned to a basic implementation of multi-threading. This took several iterations to get right. The threads were each given a section of the array to operate on and they shared a global floating point value in order to determine if the simulation is stable, much like before. Unlike before, this variables now needed to be atomic in order to avoid loss of data, as discussed previously. This slowed down the simulation drastically because atomically synchronizing a floating point variable is extremely slow. I then changed the code so that each thread would individually keep track of the stability of their own section of the grid, and when all of them agreed that the simulation was stable, they would complete. This significantly improved the multi-threaded performance, causing it to beat the single-threaded code.

Another important aspect of the multi-threading implementation was that the grid values themselves (which are shared and modified by the threads) were not atomic. This was a tradeoff I decided to make, because making them atomic would be safer, but much slower. Instead, it is possible for some threads to see stale values of cells sometimes. This is likely a significant contributing factor to why the simulation quality degreded with more and more threads. However, since most of the reads and writes to the values are simply just updating to new values, if a write is lost from the perspective of one thread it will appear as if that thread simply ran slightly faster than the others, which is possible anyway[10].

[10]There is one more detail. On x86 platforms, these shared non-atomic reads and writes are safe, but on some machines this can result in *torn-writes*. This means that a thread could possibly see half of the written value and half of the old value, which could result in nonsense numbers and could actually affect the simulation. However, because I am targeting x86 only, this is still safe.

C.1 SIMD Implementation

There were two uses of SIMD inside the solver, each for doing one iteration of a given cell under one of two situations: with a Neumann boundary condition, and without. In the case of a non-Neumann influenced cell, the cell is calculated as a sum of its nearest neighbors along with its initial value scaled by a factor of the inverse of the grid size. For successive over-relaxation, the code for an individual cell is shown in Listing 3, and an abridged SIMD version is shown in Listing 4.

```
float value = grid->values[x][y+1];
value += grid->values[x][y-1];
value += grid->values[x-1][y];
value += grid->values[x+1][y];
value += params->h * grid->initials[x][y];
value *= params->omega / 4.;
grid->values[x][y]=value+(1.f-params->omega)*grid->values[x
   ][y];
```

Listing 3: Iteration code (simplified) for a single cell. Also notice that this code uses structure-of-arrays data layout.

```c
__m256 scal = _mm256_set1_ps(params->prefix2);
__m256 res = _mm256_mul_ps(v1, scal);
grid->values[x][y] = sum8(res);
```

Listing 4: SIMD version of iteration code for a single cell. The loading of the initial vector is not shown for simplicity.

The function calls beginning with `_mm` are the SIMD instructions written in function form for C code. In this listing, `v1` is a vector that contains the nearest neighbor values along with a scaled version of `grid->initials[x][y]` and the cell's previous value. In the second line, each of these elements is multiplied by `scal`, much like multiplying a mathematical vector by a scalar value. The elements of the vector are then summed in the `sum8` function, and the result of this is stored. The two listings are equivalent. The details of the `sum8` function and how the initial vector `v1` is loaded are omitted here for brevity, but can be found in the source code.

The SIMD code that calculates Neumann boundary conditions is both more complicated and was not used in this thesis, so I will not discuss it in detail, except for one particularly interesting aspect of it which is a common pattern when writing SIMD code. As discussed previously, it is important to avoid branching in code to improve performance. The calculation for the Neumann boundary conditions involves testing a condition and throwing away a calculation depending on the result. Instead of using `if` statements, we can instead always do all of the calculations, and then at the end select the ones that we want. This is akin to transforming the code,

```
uint32_t x[2] = {123, 456};
uint32_t result[2] = {0, 0};
if(condition1) result[0] = x[0];
if(condition2) result[1] = x[1];
```

into,

```
uint32_t x[2] = {123, 456};
uint32_t result[2] = {0, 0};
result[0] = x[0] & mask1;
result[1] = x[1] & mask2;
```

where `mask1` and `mask2` are used to select which values to propagate forward in the calculation. The Intel SIMD instructions have a method to create such masks easily based on a condition (but they do it in a way that does not involve branching). Because in SIMD doing a single multiplication and doing four are no different in cost, this takes the approach of avoiding branching by doing all of the calculations even though some are optional and then selecting the ones that are desired at the end. In the SIMD code for the Neumann boundary conditions, I use this technique, and learning to do it was a particularly interesting exercise.

Software Used

1. `gnuplot`. `http://www.gnuplot.info`
2. `gcc`, and associated development toolchain. `http://gcc.gnu.org`, `https://www.gnu.org/software/binutils`, etc.
3. `python 3`. `https://www.python.org`, including packages:
   (a) `matplotlib`
   (b) `numpy`
   (c) `ctypes`
   (d) `pickle`

References

[1] Mary L Boas. *Mathematical methods in the physical sciences*, volume 2. Wiley New York, 1966.

[2] Intel Corporation. Optimization reference manual. *Intel Software Developer's Manual*, 2014.

[3] Intel Corporation. Volume 1: Basic architecture. *Intel Software Developer's Manual*, 2014.

[4] Intel Corporation. Volume 3a: System programming guide, part 1. *Intel Software Developer's Manual*, 2014.

[5] Richard J. Gonsalves. Poisson's equation and relaxation methods. `http://www.physics.buffalo.edu/phy410-505/2011/topic3/app1/index.html` accessed 2016-04-30, 2011.

[6] David Jeffrey Griffiths and Reed College. *Introduction to electrodynamics*, volume 3. Prentice Hall Upper Saddle River, NJ, 1999.

[7] IEEE / The Open Group. *gettimeofday(3) Linux Users Manual*, 2.23 edition, May 2016.

[8] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. A parallel multigrid poisson solver for fluids simulation on large grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 65–74. Eurographics Association, 2010.

[9] Marc Spiegelman. Myths and methods in modeling. *Columbia University Course Lecture Notes, available online at `http://www.ldeo.columbia.edu/~mspieg/mmm/course.pdf`, accessed May 2016*, 6:2006, 2004.