Description and Optimization of a Red-Black Multigrid Approach to Solving the Poisson Equation for a 2D Grid on Parallel Architecture

ADIV PARADISE

Submitted under the supervision of Paul Woodward to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science *cum laude* in Astrophysics.

MAY 8, 2014

Description and Optimization of a Red-Black Multigrid Approach to Solving the Poisson Equation for a 2D Grid on Parallel Architecture

Adiv Paradise

Undergraduate Thesis Report

Advisor: Paul Woodward

School of Physics and Astronomy

University of Minnesota, Minneapolis, MN 55455

parad066@umn.edu

May 4, 2014

Abstract

We describe the development and optimization of a new gravity solver. This technique solves the gravitational Poisson equation for the gravitational potential on a large uniform grid using a red-black relaxation technique, accelerated by a multigrid approach. This approach arrives at a solution in $\mathcal{O}(N)$. We further explore how this can be implemented on parallel computing platforms, as well as how it can be optimized for modern computing. Specifically, we optimize our algorithm for a 16-wide SIMD engine. We also discuss pitfalls of the algorithm, room for improvement, and next steps for further development of this algorithm. The eventual goal of this project is to adapt the algorithm for 3 dimensions for use in simulations of gravitating gaseous disks, such as might be found in spiral galaxies or protoplanetary disks.

Introduction

For decades, researchers have used computers to try to solve problems of immense difficulty and complexity that would take humans millions of years or more to solve. Even a personal computer is capable of performing in a few seconds a number of calculations that could take a human years to solve. Modern supercomputers are able to take advantage of a large number of processors running in parallel, along with the latest advances in processor design, to do even more, often performing in a matter of hours a number of calculations that would take a personal computer years, and a human eons.

One such problem is that of gravitating groups of bodies. At large scales, our universe is largely governed by gravity–it is believed to be the dominant force that causes clumps of dust and gas to coalesce into planets, stars, solar systems, galaxies, and all the way up to the superstructure of the universe. If we are to understand how it is that these bodies form, we must be able to paint a picture of that formation. Doing this requires determining the gravitational force exerted on each particle. For a large system on the scale of cosmological features such as protoplanetary disks or galaxies, this is impractical, as the sheer number of particles in something like the Milky Way is likely even greater than 10^{60}–simulating each and every particle on current computers would take uncountable lifetimes of the universe.[1] Even a computer capable of performing 10^{15} computations per second could only perform on the order of 10^{32} computations in the 14 billion years the universe has existed. Therefore, simulating large gravitating bodies is a game of approximations.

One of the most common ways to reduce the size and complexity of the problem is to approximate many small particles as one larger particle. This is called an N-Body simulation. In such a simulation, a galaxy the size of the Milky Way represented by trillions of particles is still made up of individual particles with masses several hundred times that of the Sun. A simulation of a newborn solar system involving trillions of particles and mass comparable to our own will be comprised of particles with individual masses on the order of large asteroids. Since the gravitational behavior of a concentrated mass differs greatly from that of a diffuse mass of gas and dust, these simulations must use various tricks to approximate the behavior of the nearly-continuous distribu-

tion they are modeling. This makes creating accurate simulations notoriously difficult–the success of the simulation depends on not only assuming the correct initial conditions and the correct implementation of other physical laws, but also on emergent phenomena being due to the physics of the system itself, rather than being artifacts of the various corrections and adjustments made to attempt to make a concentrated nearly-point mass behave like a diffuse mass of gas and dust. A further constraint on N-Body simulations is the speed at which the simulations can be performed. Solving for the gravity of a distribution of N bodies can be done in $\mathcal{O}(N^2)$ in the worst case,[2] though modern simulations typically accomplish it in $\mathcal{O}(N \log N)$ or better.[3;4]

An alternative approach to modelling large systems of gas and dust involves treating the system as a continuous fluid distribution–rather than being represented as discrete point masses, each differential area of the grid is assigned a density. Thus, the system can be modeled using hydrodynamic equations, and any gravity solver needs only concern itself with solving for the gravitational potential of each cell in the grid, rather than individual particles. As long as energy, momentum, and mass are conserved, modeling the average density and average gravitational potential of a cell is all that is needed–achieving the desired resolution then simply becomes a matter of using a grid with sufficiently small cells or utilizing a sufficiently accurate interpolation algorithm. This does assume that the gravitational potential is smooth. However, gravity obeys a Poisson equation,[5] such that

$$\nabla^2 \phi = 4\pi G \rho \tag{1}$$

where ϕ is the potential at the given point, G is the gravitational constant, and ρ is the density at the given point. Thus, since the second derivatives of the potential must exist and must be real, it's a fairly good assumption that the potential is smooth. In fact, this is a problem that N-Body simulations must deal with–discrete point masses represent gravitational singularities that must be "softened" to maintain accuracy.

Solving the gravitational Poisson equation for a continuous distribution of gas and dust has many advantages, beyond the simple fact that a continuous fluid is qualitatively more similar to a continous distribution of gas and dust than is a distribution of massive discrete particles. Many

other Poisson-like equations are often solved for fluids, such as the Dirichlet equation for temperature,[6]

$$\nabla^2 T = 0$$

where T is the temperature, or the Poisson equation for the scalar electric potential,[7] given by

$$\nabla^2 \varphi = \frac{-\rho}{\epsilon}$$

where φ is the scalar potential, ρ is the charge density, and ϵ is a permittivity constant. This means that researchers have been investigating ways to efficiently solve these types of equations for years. One such scheme is called a red-black relaxation algorithm. This algorithm divides the grid into a checkerboard to perform half as many computations to reach the same level of iteration.[8] When accelerated with a multigrid component, wherein the residual error for a solution to the grid is itself relaxed on a coarser grid into a correction factor and then applied to the original fine grid,[9] this scheme can solve Poisson equations in $\mathcal{O}(N)$.[8]

Red-Black Relaxation

The checkerboard pattern of the red-black relaxation algorithm can be understood as being an intrinsic property of Poisson equations. **Eq. 1** can be rewritten as

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\phi = 4\pi G\rho \tag{2}$$

We omit cross-terms such as $\partial^2\phi/\partial x\partial y$ because gravitational force can be represented by orthogonal components, so we do not expect the potential to have any cross-axis dependence. At the long-time limit, or the limit in which the solution has relaxed and converged and, barring disturbance of the system, ϕ is constant, we can approximate $\frac{\partial^2\phi}{\partial x^2}$ as

$$\frac{\partial^2\phi}{\partial x^2} = \frac{1}{\Delta x^2}(\langle\phi\rangle_R - \langle\phi\rangle + \langle\phi\rangle_L - \langle\phi\rangle) \tag{3}$$

where we understand Δx^2 to be suitably small, and we assume Δx to be the width of a grid cell in both the x and y axes.[8] $\langle\phi\rangle_R$ and $\langle\phi\rangle_L$ are the average potential values in the cells to the right and left respectively of the current cell being solved, represented as $\langle\phi\rangle$. Thus, we can express $\nabla^2\phi$ as

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\phi = \frac{1}{\Delta x^2}(\langle\phi\rangle_R + \langle\phi\rangle_L + \langle\phi\rangle_U + \langle\phi\rangle_D - 4\langle\phi\rangle) \tag{4}$$

where, as before, $\langle\phi\rangle_U$ and $\langle\phi\rangle_D$ are the average potential values in the cells above and below the current cell. Therefore, taking **Eq. 2**, we can express the average potential within the current cell as

$$\langle\phi\rangle = \frac{1}{4}(\langle\phi\rangle_R + \langle\phi\rangle_L + \langle\phi\rangle_U + \langle\phi\rangle_D) - \pi G\rho\Delta x^2 \tag{5}$$

where ρ is the average density within the cell. **Eq. 5** forms the basis for the red-black algorithm. The solution to $\langle\phi\rangle$ for a given cell depends only the cells to the North, South, East, and West–there is no dependence on the 4 cells diagonally adjacent to it. This means that on a grid laid out like a checkerboard, with red and black cells, each red cell depends only on black cells, and vice versa.

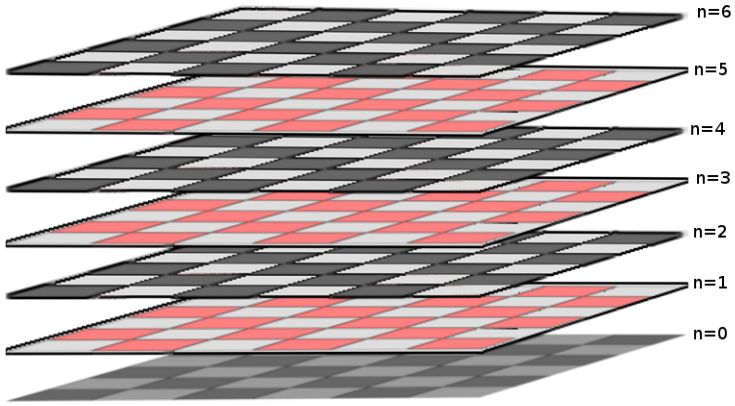

Figure 1: In a red-black relaxation algorithm, at each iteration level the cells of the previous level are leapfrogged by the other cells, resulting in alternating levels of red and black. In this way, a given iteration level n can be reached in half the work that would be required if the entire grid were computed for each iteration level.

Therefore, if we consider each iteration to represent a certain depth towards a solution, then reaching iteration level 1 only requires that we compute half the cells in the grid.[6] We can then use

those cells to compute the other half, which will then be at iteration level 2. In the same number of computations that would have been necessary to compute the entire grid to iteration level 1, we have reached iteration level 2. Thus, the red-black technique allows us to alternate between red and black relaxations, with the red and black cells leapfrogging each other, as shown in **Fig. 1**. We are able to reach iteration level n in half the work normally required. Each iteration requires only one computation per cell, meaning the work required to solve the grid scales with N, the number of cells in the grid.[8]

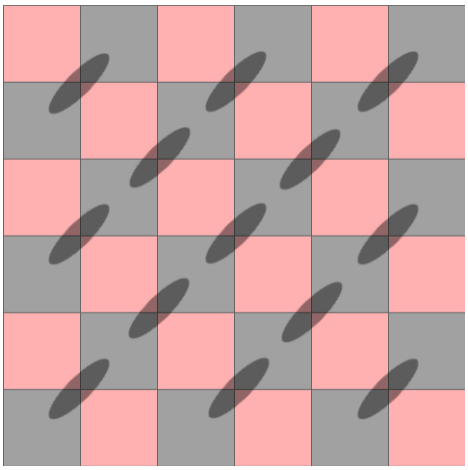

Figure 2: Since diagonally-adjacent red cells share two of their neighbors, the computation for $\langle\phi\rangle$ can be split up into a sum of two other sums of adjacent neighbors, as $((\langle\phi\rangle_L + \langle\phi\rangle_U) + (\langle\phi\rangle_D + \langle\phi\rangle_R))$. This way, each pair-sum only has to be computed once, and can be reused for the other cell that depends on it.

We can also save some work by noting that diagonally adjacent red cells share two black neighbors, as shown in **Fig. 2**. This can be leveraged by rewriting **Eq. 5** as a sum of two pair-sums:

$$\langle\phi\rangle = \frac{1}{4}\big[(\langle\phi\rangle_L + \langle\phi\rangle_U) + (\langle\phi\rangle_D + \langle\phi\rangle_R)\big] - \pi G\rho\Delta x^2 \tag{6}$$

Thus, each of the two pair-sums can be computed once, and then reused for a second cell.[10] For example, the pair-sum $(\langle\phi\rangle_L + \langle\phi\rangle_U)$ is also used for the cell's Northwest red neighbor. This allows for a further reduction in the number of computations necessary for each iteration. This illustrates some of the main benefits of the red-black technique–its inherent structure exposes sev-

eral redundancies that are easy to optimize, and it presents many opportunities for vectorization and parallelization.[6]

Multigrid Acceleration

The red-black relaxation technique detailed above can work fairly well for arriving at a solution, but it does suffer from the drawback that since at any given point, each cell's value depends only on the value of its neighbors, information only travels one cell per iteration.[9] On a large grid, this means that arriving at a relaxed, convergent solution can take a great deal of iterations, pushing the number of computations required to reach a solution from $\mathcal{O}(N)$ to $\mathcal{O}(N^{3/2})$. It is, however, possible to accelerate that information transfer by utilizing a multigrid technique, wherein several different grids of coarser resolutions are used to compute correction factors to wipe out low-frequency errors in the grid.[9]

To compute these correction factors, we first need to know how far from the solution each point is. To determine this error, we define a residual for each cell such that[11]

$$\langle R \rangle = (\langle \phi \rangle_R + \langle \phi \rangle_L + \langle \phi \rangle_U + \langle \phi \rangle_D) - 4\pi G \rho \Delta x^2 - 4\langle \phi \rangle \tag{7}$$

Thus, the residual is simply proportional to the difference between the value of the cell and what the value would be if the cell satisfied the Poisson equation. If the equation is satisfied, the residual is 0. One can then coarsen this residual. This residual then takes on the role of the source term in a new relaxation scheme, given by

$$\langle C \rangle = \frac{1}{4}(\langle C \rangle_R + \langle C \rangle_L + \langle C \rangle_U + \langle C \rangle_D) - \frac{1}{4}\langle R \rangle \tag{8}$$

where $\langle C \rangle$ is a scalar correction factor. When $\langle C \rangle$ is relaxed in the same way that $\langle \phi \rangle$ is relaxed, then $\langle C \rangle$ can be used as a correction factor for $\langle \phi \rangle$. Once $\langle C \rangle$ is interpolated down to the resolution

of the fine grid, it can simply be subtracted from $\langle\phi\rangle$, such that

$$\langle\phi - C\rangle = \frac{1}{4}(\langle\phi - C\rangle_R + \langle\phi - C\rangle_L + \langle\phi - C\rangle_U + \langle\phi - C\rangle_D) - \pi G\rho\Delta x^2 \tag{9}$$

Note then that $\langle\phi - C\rangle$ is a solution to the gravitational Poisson equation.[11] Furthermore, since the correction factor C is computed on a coarser grid than the original fine grid, not only does it require fewer computations, but due to the larger grid cells, information travels farther across the grid with the same number of iterations.[9] If the coarse correction is not relaxed fully, a residual can be computed for the coarse grid in the same manner as for a fine grid. This residual can then be coarsened even further, and a correction to the correction computed on a grid now 16 times smaller than the original fine grid. Thus, in order to arrive at a suitable solution on the fine grid, an algorithm may move up and down through the grid levels, relaxing solutions through red-black techniques, applying corrections, relaxing the solutions a bit closer to the solution, computing new residuals, and so on. This can greatly accelerate the process of finding a solution to the Poisson equation, allowing for solutions to be found with $\mathcal{O}(N)$ computations.[9] The time it takes to perform those computations can be reduced significantly by taking advantage of parallel processing and memory-optimized data management.

Parallelization and Optimization

While in theory, this red-black multigrid approach seems like it would be an incredibly fast and efficient algorithm for computing the gravitational potential on a uniform grid, when it comes to performing the algorithm on a parallel supercomputing architecture there are many aspects that conspire to complicate the implementation of an efficient red-black multigrid algorithm. These include obstacles involving efficient use of the SIMD engine, taking advantage of cache sizes on chips and the sizes of cache lines that the computer uses when reading memory, accomodating multiple threads working on the same problem, and avoiding data unalignments.[10]

The SIMD engine of a processor is a mechanism by which the processor is able to take a set

of similar instructions, or an instruction that needs to be performed many times on a chunk of data, and perform a certain number of those instructions simultaneously.[12] It is the equivalent of being presented with two apples, each of which needs to go into its own bucket, where the two buckets are nearly identical and are arranged next to each other. A human with two hands is able to pick up both apples and place each in its bucket simultaneously. A SIMD engine is capable of the same thing, but often rather than doing 2 things at once, it can perform 4, 8, 16, or even 32 operations at once. In order to take advantage of this feature, algorithms must be written in such a way that allows the processor to interpret the maximum number of sets of commands as vector operations that could be performed by the SIMD engine. When working with something like a two-dimensional grid, this can affect how much of the grid is processed at once. SIMD engines have limitations, such as an inability to vectorize exceedingly complex instructions. As such, it is in the programmer's interest to keep instruction sets as linear as possible.

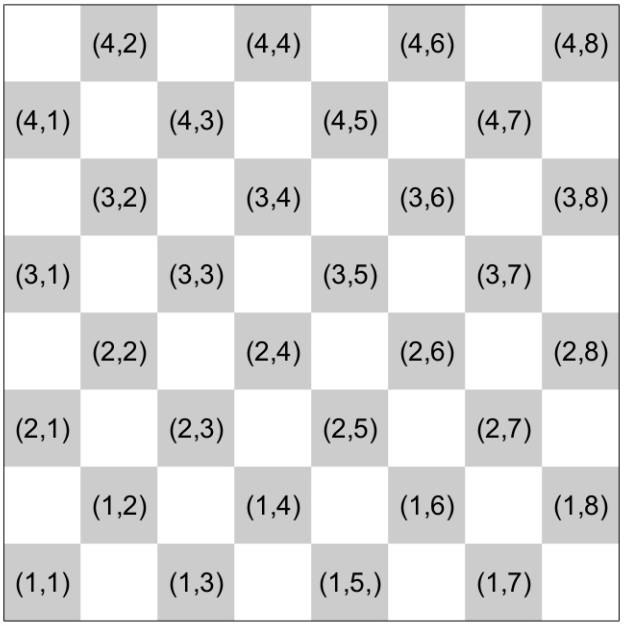

Figure 3: When computing the pair-sums, one would like to avoid cases where cell (i,*) is being added to cell (i+1,*). However, taking a square section of the checkerboard, at least half the pair-sums will involve such a summation. Furthermore, because pair-sums are shared, as shown in **Fig. 2**, many relaxation computations will involve unaligned data and indexing that does not allow for SIMD optimization, as one pair-sum may be added to another pair-sum separated from it in memory by more than a cache line, requiring a second fetch.

It's also important to avoid issues with data alignment. This issue primarily arises with the use of pair-sums. Unaligned data can mean that the compiler won't recognize the operation as vectorizable, or it can mean that in order to satisfy the requirements of the operation, the processor will have retrieve an additional cache line of data just to provide one more cell.[13] Making an additional access to memory can be extremely costly, as a memory access is orders of magnitude slower than addition or multiplication.[10] **Fig. 3** shows numbered black cells on a grid. Data is unaligned when a cell (i,*) must be compared to a cell (i+1,*). This may be a problem for the SIMD engine for the computation of the pair-sums themselves, but when it comes time to add the pair-sums and compute the actual relaxation, some pair-sums may be added to other pair-sums separated in memory by more than a cache line. The natural configuration for a section of grid to be computed is a square or rectangle, as data is organized in columns and rows, but this particular cut of a checkerboard pattern means there will be a great deal of data unalignment. There are various tricks one can employ to attempt to reduce the number of data unalignments, but in this configuration, it is impossible to fully avoid data unalignment.[10] One way that could avoid this problem would be to rearrange the necessary cells and pair-sums into linear arrays that have been pre-shuffled to allow the thread to perform all the necessary computations, but without any data unalignment. However, the thread would quickly run out of operations to perform before needing to fetch new data from memory. The effect would be that rather than zooming down the linearized array of data, the thread would start and stop, spending most of its time stopped, waiting for new data.[10]

One would also like to allow multiple threads to work on the grid at the same time. If one thread can accomplish it decently quickly, then several threads can do it even faster. However, this requires that the grid be split up into segments for each thread to work on. This poses a problem, because a red-black algorithm cannot compute many iterations near a boundary. After the second iteration, the first row in from a boundary must be considered corrupted, because it will now consist of a relaxation that mixed cells at iteration level 1 from inside the grid with boundary cells, who have not been changed, and thus are still at iteration level 0. For a datapoint to be uncorrupted,

all progenitor cells must be at the same iteration level. And furthermore, on the third iteration, the second row in has been corrupted, as now the cells in the second row have been produced with corrupt data values. So with each iteration, the point at which data can be trusted moves farther inward.[6]

With one thread, this isn't necessarily a problem–one starts with a larger grid than is necessary and throws away the corrupted edges at the end. However, if multiple threads are working on the problem, and the grid is split into sections, there needs to be a way to ensure continuous information transfer at the boundaries of each section to avoid the formation of visible seams. One way to do this is for each thread to frequently communicate to neighboring threads what its edge values are. Each thread can then use the edge information it receives from other threads to compute relaxations along the relevant boundaries. However, this requires stopping frequently to pass messages, which can be time-consuming and costly.[10] A faster approach would be to allow threads to overlap their sections of the grid, such that the innermost uncorrupted cells computed by one thread coincide with the first row of corrupted cells computed by the neighboring thread. This would allow a thread to proceed down the grid, performing several iterations, without ever having to wait for a neighboring thread. However, this increases the amount of redundant work being done.

Furthermore, allowing overlap can cause problems in conjunction with accomodations made for the SIMD engine. A natural way to accomodate the SIMD engine is to work with a "briquette" of data, representing a square chunk of the grid whose dimensions are the width of the SIMD engine.[10] For a 4-wide SIMD engine, this would imply a 4-by-4 square. For a 16-wide engine, this would imply a 16-by-16 square. This way, the SIMD engine can compute entire columns of the briquette at once. However, if threads are allowed to overlap, then rather than repeating on the basis of a multiple of 2, such as one thread every 16 cells, or every 8 cells, etc,threads will repeat on a basis that is not easily divisible by powers of 2, nor will it easily go into a power of 2. For example, if a simulation uses a 16-wide SIMD engine, and loses 2 cells at each boundary, then to avoid seams, the next thread over will have to be offset from the first not by 16 cells, but by 14

cells. 14 factors into 2 and 7–while the 2 can go into other factors of 2, that 7 will never divide cleanly. The exact periodicity of a simulation's sweeps depends on the width of the SIMD engine (and thus the grid chunk being operated on), the size of the grid, and the degree of overlap between threads. This poses problems for ensuring that the entire grid is covered, as the programmer must ensure that enough sweeps are performed on all grid resolutions involved in the algorithm.

Paradoxically, working in the context of a fixed-size briquette sized for optimal SIMD vectorization and memory use can also reduce the efficiency of the SIMD engine. The SIMD engine will always proceed at the same speed, regardless of it's doing the maximum number of things possible. So if a 16-wide SIMD engine is given 15 things to do, it will only do 15 things in the same amount of time that it could have done 16–such a usage represents an inefficiency.[13] Working with a briquette with the same dimensions as the SIMD engine, all of the SIMD engine's registers will be used to assemble the briquette and compute the first set of pair-sums. However, the first relaxation operation will necessarily have fewer operations in each column than the pair-sum operation. Assuming the fast-running index traverses the columns of the briquette, and thus the SIMD engine computes entire columns at a time, each cycle of the SIMD engine will perform less than the maximum number of operations. And the same will be true for the next set of pair-sums, which only depend on the cells that were just relaxed, and even fewer operations for the next set of relaxations, etc–for much of the algorithm executed for each briquette, the full width of the SIMD processor is not in use. This could be addressed by arranging the data into linearized arrays, as discussed earlier, but even if the problem of memory fetching were avoided, there would still be a rearrangement cost to put the data in those arrays that would outweigh the loss of potential efficiency in the usage of the SIMD engine.[10] Therefore, a fast red-black multigrid algorithm will need to find a way to deal with these obstacles.

Implementation

For our implementation, we anticipate an architecture that uses a 16-wide SIMD engine.[10] Thus, the natural dimension for our briquette is 16 cells. However, we wish to avoid the various issues detailed above. In particular, we wish to address the data alignment problems and non-uniformity that arises from horizontal and vertical cuts, as shown in **Fig. 3**. One of the pecularities of a checkerboard pattern is that there is a strong diagonal bias–a checkerboard is essentially two grids, one red and one black, which have been rotated 45° and overlaid on top of each other, and then cropped to fit in a smaller grid in the original coordinate system. This diagonal bias is further evident in **Fig. 2**–all the pair-sums are diagonally oriented, as are the relaxation computations. Therefore, we will use a briquette that is also diagonally oriented, as shown in **Fig. 4**.

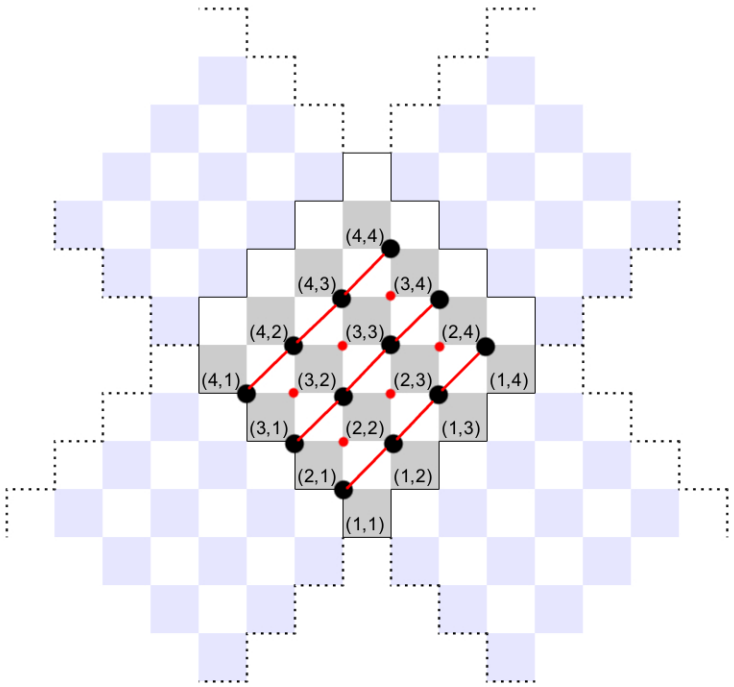

Figure 4: We use a briquette pattern oriented along a rotated coordinate system, as indicated by the cell indexing in the figure. Black dots correspond to vertical pair-sums of black cells, and red dots correspond to vertical pair-sums of red cells. The red lines indicate that relaxations involve summing horizontally-adjacent pair-sums. This scheme ensures proper data alignment, as all pair-sums are confined within their column, and all relaxations involve summing pair-sums indexed with identical (i,*) patterns.

Using a diagonal briquette eliminates the issue of data alignment. To perform the pair-summations, only cells within the same column are added–the operation involves adding cells (i,a) and (i+1,a), where a is the column number, and i is along the fast-running index. The relaxations themselves involve adding horizontally adjacent pair-sums–the operation involves adding sums (b,i) and (b,i+1) where b is the row number, and i and i+1 represent the column number. This sort of operation is easily vectorized for use by a SIMD engine.[12] Furthermore, this layout exhibits uniformity–each "column" has the same number of black or red cells, meaning the thread traversing the grid does not need to know how many briquettes have passed since the beginning of the sweep.

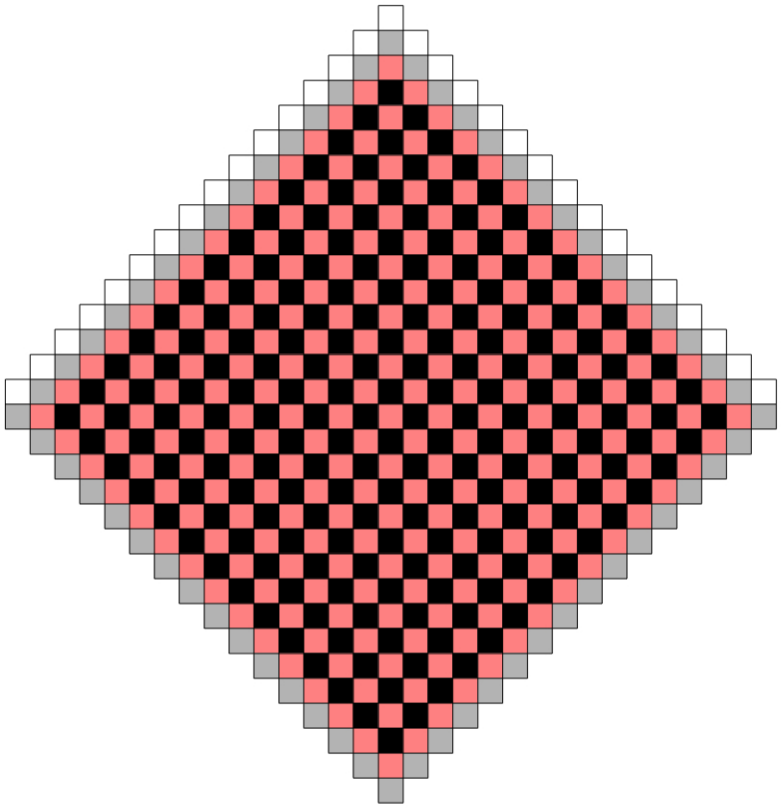

Figure 5: The briquette used in our algorithm. Upon each relaxation, a row or column of cells is lost from each edge. The first red cells computed occupy a square area that is 15 by 15 in our example, as opposed to 16 by 16. We don't lose a row of reds on the bottom or left edge, as the cells on those edges are black cells. The first set of relaxed black cells is thus 14 by 14. Relaxed black cells are shaded in black, and relaxed red cells are shaded red. Unrelaxed cells are gray or white. The red cells are one iteration level further than the base gray cells, while the black cells are two iteration levels further.

Our diagonal briquette involves a 16 by 16 diagonal square of black cells as well as a 16 by 16 diagonal square of red cells. To minimize the amount of overlap required between threads, for each sweep we perform only one red-black iteration, bringing the relaxed red cells up one level of iteration, and bringing the relaxed black cells up two levels of iteration, as shown in **Fig. 5**. This means that two rows are lost, so as one moves along the grid in a rotated-vertical direction, a new briquette will begin every 14 cells, as shown in **Fig. 6**. Since the pair-sums are computed vertically, all the pair-sums on the leading edge of the briquette can be reused to compute the margin between the current briquette and the next briquette. Thus, every briquette after the first in a sweep will involve relaxing 16 columns of black cells, as opposed to the 14 relaxed in the first briquette. These black cells establish a trailing edge behind the very front of the briquette, as those 16 columns are offset from the 16 of the base state by 2 columns.

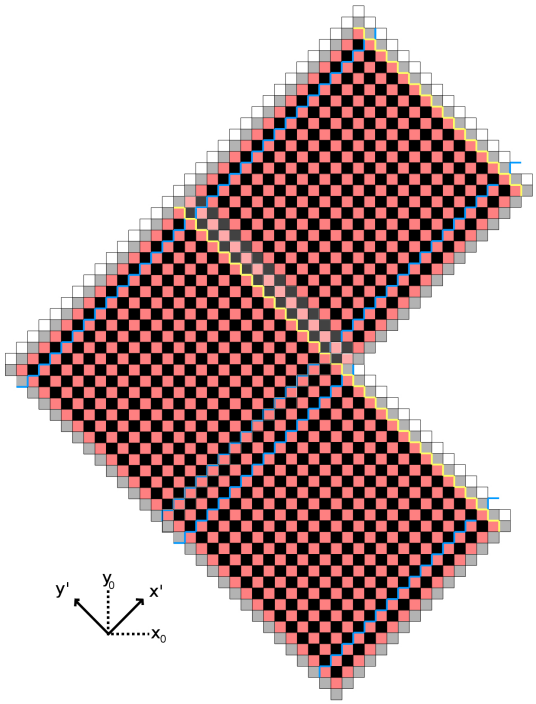

Figure 6: Vertically adjacent briquettes will overlap by 2 cells, as shown by the margins bounded by blue lines. This overlap allows for a seamless relaxed grid without relying on message passing between threads. The horizontally adjacent briquette, or the one to be computed after the current one, is situated directly adjacent, with no overlap. The pair-sums and values from the leading edge of the previous briquette are reused to compute the trailing edge of the new briquette, creating a seamless row down which the thread moves.

In order to assemble these rotated briquettes, one can either rotate the coordinate system each time the briquette is assembled, or the entire grid can be rotated at the onset of the algorithm. The latter saves quite a bit of computation time. However, this rotation would still be computationally slow if done one cell at a time. The challenge then is to optimize it. To take advantage of the way memory is read in units of cache lines,[10] we rotate the grid in diagonal strips running from the southeast corner to the northwest corner–southeast is to become the rotated south, and northwest is to become the rotated north. To form each strip, 16 cells are pulled from memory–a horizontal strip comprised of 8 red and 8 black cells. Each cell is then dumped into a diagonal in the rotated strip. This is repeated with a new strip of 16 cells offset up one row and shifted one cell to the left, as shown in **Fig. 7**.

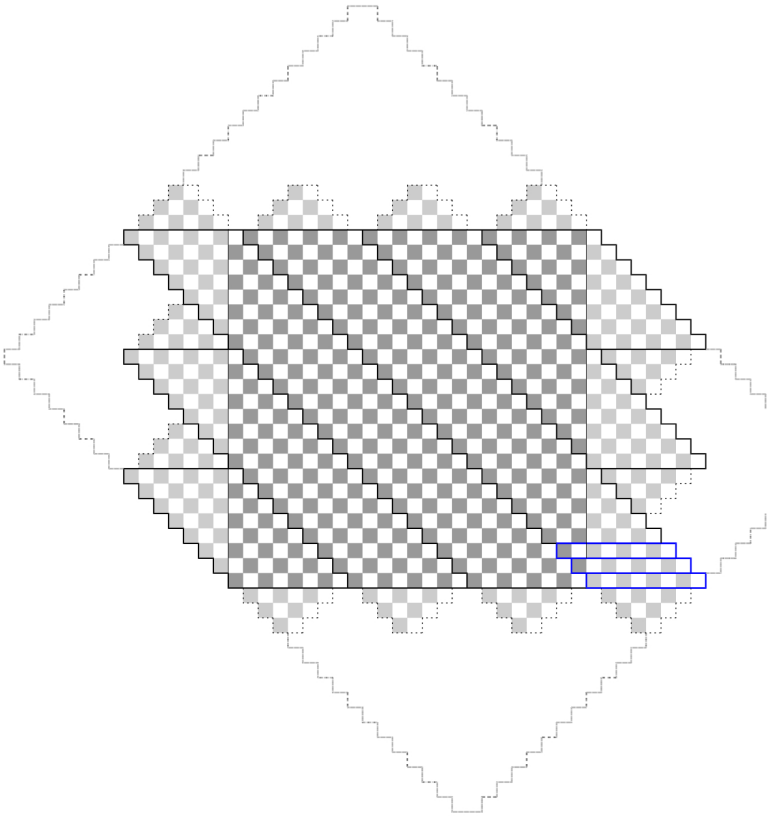

Figure 7: In order to rotate the grid, we rotate individual strips by pulling 16-cell (in this case 8-cell–this is an 8-wide example to save space) horizontal strips from the original grid, and placing them in the diagonalized strips. Once the rotation is complete, these strips make up the interior of a somewhat larger, rotated grid.

Once the grid has been rotated, threads can traverse it horizontally, relaxing the grid and converging toward a solution. However, threads that are farther from the central latitude of the rotated grid will traverse far less of the actual grid than will threads at central latitudes. This represents an enormous waste of computing time and power. This can be avoided by rearranging the grid. To do this, we slice the grid down the central latitude, such that we have a section of the grid that includes an upright triangle comprising half the original grid, and another section that includes an upside-down triangle comprising the other half. These triangles can be stacked on top of each other, forming a parallelogram.[10] To take matters even further, this parallelogram can be shifted left, creating a left-justified edge. The right half of the new grid is not shifted uniformly in order to create a right-justified edge. This leads to some rows having a gap roughly the size of a briquette between the row from the top half and the complementary row from the bottom half, as shown in **Fig. 8**.

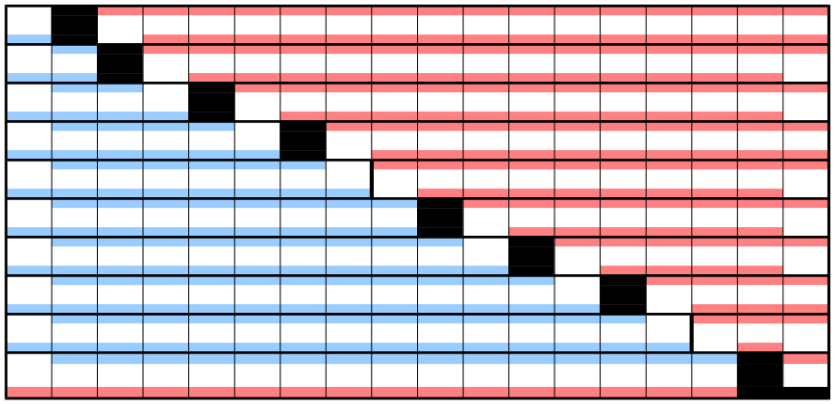

Figure 8: To maximize the efficiency with which the rotated grid is computed, we split the rotated original grid into two halves–a top half (blue) and a bottom half (red). We then lay the bottom half neatly on top of the top half, forming a parallelogram. This parallelogram is then shifted left to create a left-justified edge. Not every row from the lower half is shifted all the way over to create a right-justified edge as well. This leaves briquette-sized gaps in many rows, represented here by large black squares. Because most rows have been offset, the overlap regions have in many cases been offset by a briquette's width. This is indicated by the colored strips along the bottoms and tops of each row. Once a briquette has been assembled, those colored strips should line up perfectly. Some rows from the bottom half, due to the quirks of creating a right-justified edge as well, do not have offset overlap regions. Whether or not this is the case is controlled by flags.

Thus, we arrive at a rearranged grid where each thread has an approximately equal amount of work to do, and the majority of the work corresponds to cells from the original grid. However, because we have shifted several of the rows to create our justified edges, the overlaps have become offset. For example, the left rows, corresponding to the top half of the rotated grid, have their bottom 2 rows come from the row below. In each case, these 2 rows are shifted one briquette's worth to the right. For the rows on the right, in most cases the overlap is taken from the top, and the top two rows have been shifted one briquette left. However, because the offset is not entirely consistent in order to enforce a right-justified edge, some rows do not move with respect to their neighbors, and the top two rows are not offset. Therefore, when doing the rotation, the algorithm takes note of which rows will have this property, and sets a series of flags. This rearrangement is performed once at the start of each grid–not at the start of each sweep or iteration, but rather each time work is begun on a different grid than what was previously in use. This way, we are able to create an environment in which the SIMD engine is able to vectorize most operations, although we do assume an inevitable cost in efficiency of not being able to always use all 16 registers in the SIMD engine. We avoid data unalignments, manage to accomplish a suitable amount of work before needing to make another memory access, access the memory in a way that takes full advantage of the cache lines to make sure we don't make more memory calls than necessary, and we avoid the inconsistencies and irregularities that pop up when traversing a checkerboard grid in the traditional horizontal and vertical dimensions.

Results and Future Work

This particular implementation has not yet been timed. However, it is written to find a solution for the gravitational potential in the least amount of work possible, and to do that work in the most efficient way possible. The algorithm is able to produce gravity wells on the fine grid, as well as compute the residual and relax the coarse grid corrections on several coarse grids. Our test runs used units of Neptune masses and Megameters, with a fine grid cell width of 10 Megameters. We

ran the algorithm on a 256 by 256 grid, performing 8 iterations on the fine grid, 4 iterations on the first coarse grid, and 2 iterations on the second coarse grid. This leads to a loss of 48 cells from each edge, so if one were to solve a very large grid by breaking the grid into smaller, 256 by 256 squares and solving those squares,[14] they would have to overlap their neighboring threads by 48 cells. The density distribution used was a simple gaussian. **Figures 9**, **10**, **11**, and **12** show the results of these test runs.

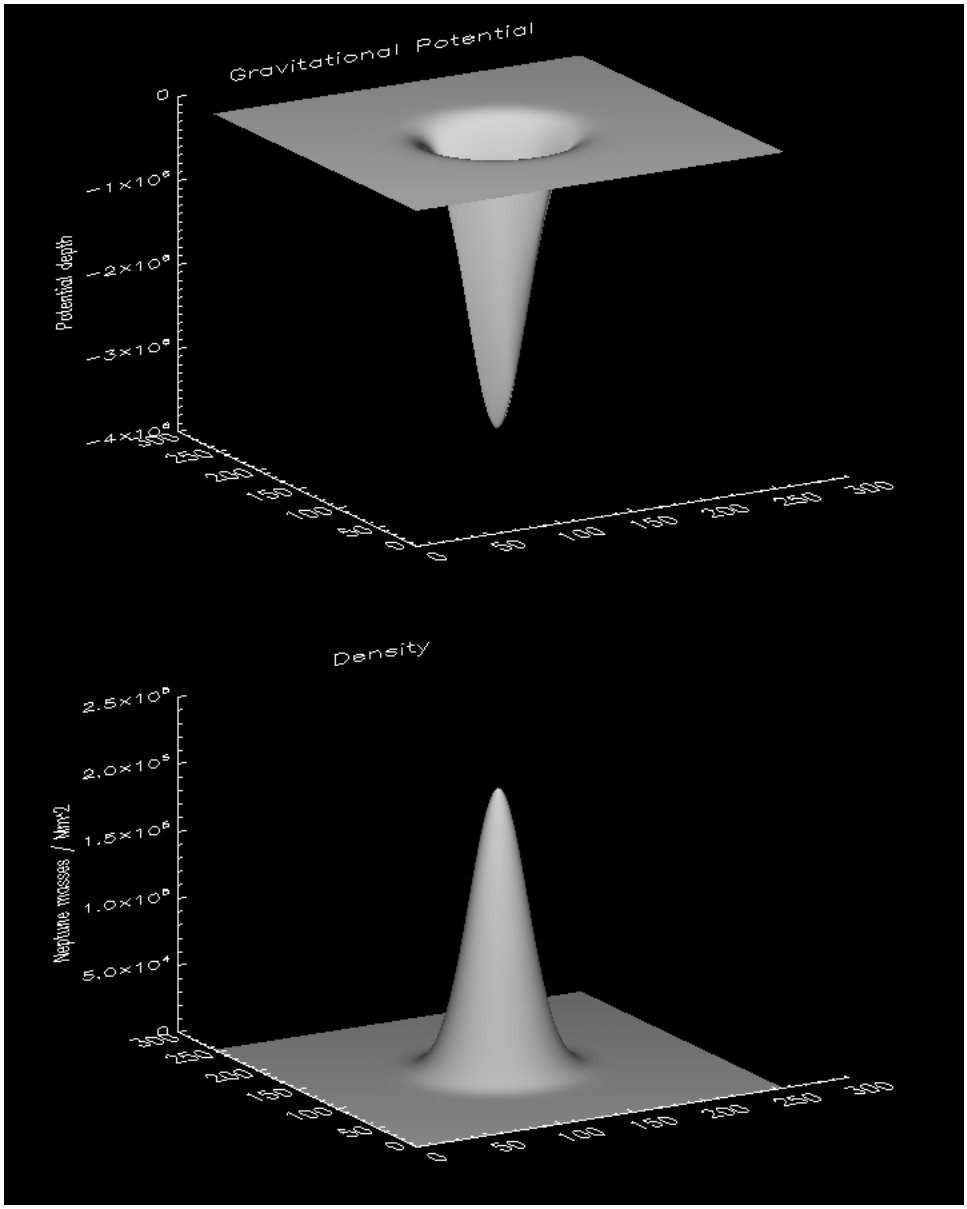

Figure 9: The gravity well for a gaussian density distribution after 8 iterations.

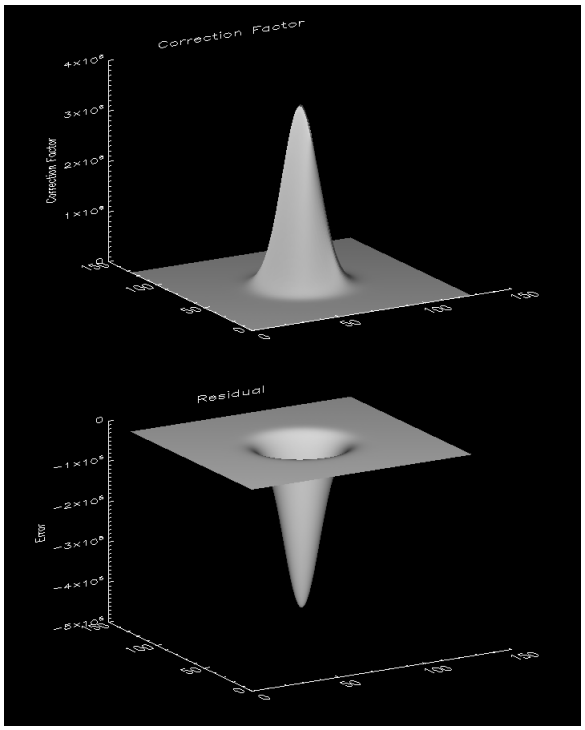

Figure 10: The first coarse-grid correction and the residual from the initial fine grid solution.

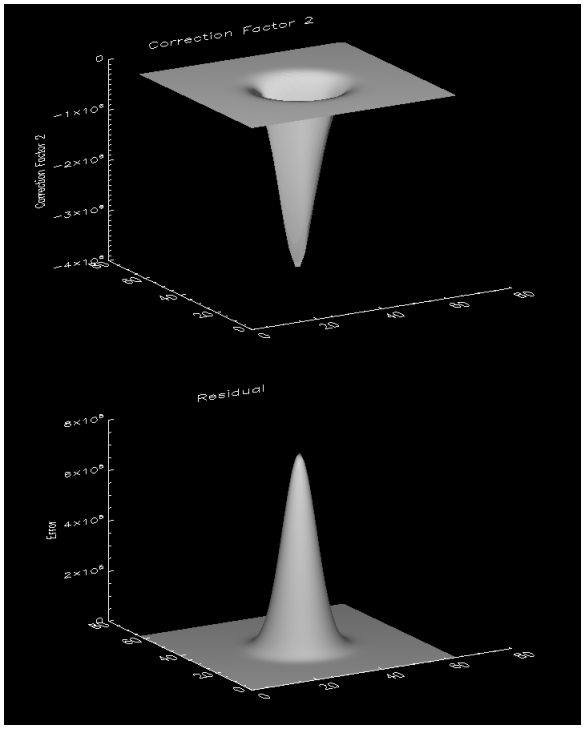

Figure 11: The second coarse-grid correction and the residual from the first coarse-grid correction.

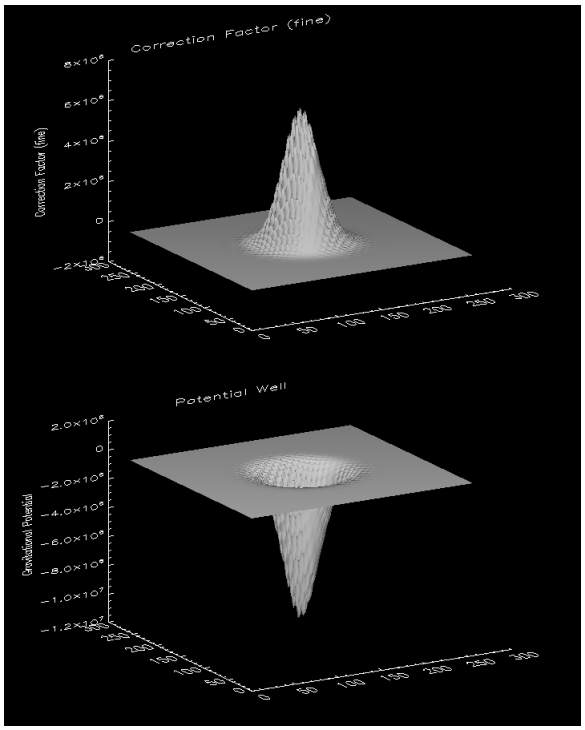

Figure 12: The gravity well for a gaussian density distribution after both correction factors have been interpolated down and applied. Note the dimpling. It is believed that this is due to interpolation when the system has not completely converged, such that red and black are at different depths of relaxation, leading to a very fine dimpled pattern. This is seized upon and extrapolated by interpolation routines.

After the first interpolation, a sort of dimpling becomes apparent. With each interpolation, it worsens, such that when the two coarse grids have been applied to the fine grid, it has a distinctly dimpled texture, as seen in **Fig. 12**. This is most likely due to the fact that since, as part of the red-black technique, the red and black cells leapfrog each other in terms of depth of iteration, when the grid has not completely converged on a solution, and the red cells were the last to be computed, the red cells will have a systematically higher or lower value than the black cells. This will result in very high-frequency dimpling–every cell is a peak or a valley, as seen in **Fig. 13**. One possible way to counteract this is to destroy the information about the dimpling. By integrating up to a coarser grid, the dimpling is averaged out, and that information destroyed. If the grid is then interpolated back down to the original fine resolution, then, assuming the potential is a smooth function, the grid should take on the same approximate shape it had before the integration, but

minus the dimpling. This will be an area of further research. Another goal for further research is to adapt this algorithm for 3 spatial dimensions. Doing so would allow for very accurate simulations of gravitating galaxies and protoplanetary disks.

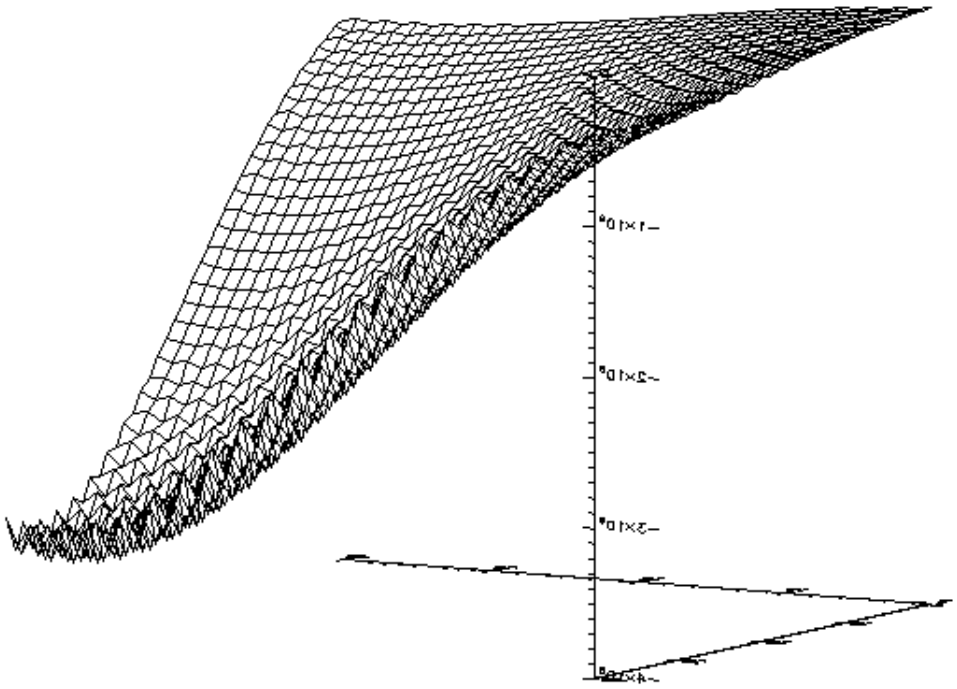

Figure 13: This slice of the original gravity well, as seen in **Fig. 9**, shows that even at that point, dimpling exists at a fine level. It is very clearly every other cell, in a checkerboard pattern, suggesting that either the red cells or the black cells are systematically lower than the others. This could potentially be solved by smoothing each grid after each final iteration.

Conclusion

The red-black technique, coupled with a multigrid acceleration, can be a very fast and very powerful tool for solving Poisson-type equations on uniform grids. However, doing so optimally and efficiently can pose major challenges. We explore an algorithm that takes advantage of the naturally-rotated coordinate system of a checkerboard pattern to bypass many of the obstacles that would otherwise impede such an algorithm. We further use a grid rearrangement to repack the data into a dense format that maximizes the productivity of each thread, allowing this algorithm to

run efficiently on parallel architectures. This is further aided by the relative independence of each thread–no messages need to be passed, and the only requirement is that each thread's neighbors must have begun their previous iteration before the thread in question begins its next iteration. However, without smoothing, this algorithm may not yield satisfactorily accurate results unless the grid has been relaxed to a converged solution. Once the dimpling problem is solved, the algorithm should be expanded to 3 spatial dimensions. Once the algorithm is fully functional and has been shown to be satisfactorily fast in both 2 and 3 spatial dimensions, this algorithm could have wide application beyond just gravity. As noted in the introduction, many other phenomena obey similar equations, such as the scalar electric field potential, temperature, and other diffusive phenomena.

Acknowledgements

This work could not have been possible with out the outstanding help, support, instruction, and mentorship of Dr. Paul Woodward. The hours upon hours that he contributed to this project were instrumental in its completion. Thanks are also extended to the Laboratory of Computational Science and Engineering, for providing funding, office space, and computing power. Thanks are given to Mike Knox for his generous help learning to use the computing resources at LCSE. Thanks are given as well to the friends and family of the author, who lent support, encouragement, and interest for the duration of this project.

References

[1] C. S. Kochanek, The Astrophysical Journal **457**, 228 (1996).

[2] M. S. Warren and J. K. Salmon, in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing* (1992).

[3] W. Dehnen, Journal of Computational Physics **179**, 27 (2002).

[4] J. Barnes and P. Hut, Nature **324**, 446 (1986).

[5] B. Carroll and D. Ostlie, *An Introduction to Modern Astrophysics* (Addison-Wesley, 2007), 2nd ed.

[6] P. Woodward, *2d heat diffusion* (2005), powerpoint.

[7] D. J. Griffiths, *Introduction to Electrodynamics* (Pearson, 2013), 4th ed.

[8] P. Woodward, *Notes for assignment 5, relaxation methods*, computational Methods in the Physical Sciences.

[9] P. Woodward, *Notes for assignment 6, multigrid relaxation*, computational Methods in the Physical Sciences.

[10] P. Woodward, Private communication.

[11] P. Woodward, *Thoughts on multigrid gravity solve*.

[12] *Basics of SIMD Programming*, The Linux Kernel Organization (????).

[13] C. Piper, *An Introduction to Vectorization with the Intel Fortran Compiler*, Intel (2012).

[14] P. Woodward, *Poisson rumination*.