Simulating a dark matter halo using a parallel grid method

Otto Hannuksela *Janne Lampilahti*

May 31, 2015

1 INTRODUCTION

Dark matter appears to be the dominant form of matter in the universe. While no direct observations of dark matter exist, it has been observed indirectly through its gravitational interaction with visible matter and radiation (Roos, 2012). The effect of dark matter on structure formation in the universe is the subject of an increasing scientific interest and a useful tool in the search for possible candidates of dark matter. Current progress in the investigation of structure formation is mainly driven by advances in computational methods and capabilities (Kuhlen et al., 2012).

In this work we simulate a dark matter halo, neglecting baryonic matter, by using a particle based method and solving the Poisson's equation of gravity. The software used is *Uintah* (available from `http://uintah.utah.edu/` during the writing of this report), a parallel grid framework with a support for particle interactions and adaptive mesh refinement and which is aimed at solving partial differential equations. A major part of the project was to learn how to use the tools provided by Uintah.

2 METHODS

In this section the relevant theory and numerical methods are described. The method in short was to use a Poisson solver operating on a grid and an N-body integrator operating on particles, that are coupled via interpolation between the grid and the particles. This method is used in the context of massive collisionless particles interacting through gravity (e.g. Hockeny and

Eastwood, 1985).

2.1 Theory

A distribution of mass density ρ gives rise to a gravitational potential ϕ according to the Poisson's equation of gravity

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

where $G \approx 6.674 \times 10^{-11}$ Nm^2/kg^2 is the gravitational constant. In our simulation we normalize $G = 1$. The corresponding force field can be solved from the gradient

$$\mathbf{F} = -\nabla\phi. \tag{2}$$

In our simulation a distribution of massive particles create the mass density. A new position and velocity for the particles after a time step dt can be solved from the Newton's equation of motion.

$$\mathbf{v}(t+dt) = \int_t^{t+dt} \frac{\mathbf{F}(t')}{m} dt' + \mathbf{v}(t) \tag{3}$$

$$\mathbf{x}(t+dt) = \int_t^{t+dt} \mathbf{v}(t')dt' + \mathbf{x}(t) \tag{4}$$

The use of this theory assumes that we do not have relativistic speeds or masses and that the maximum grid size is small enough that expansion of the universe can be neglected.

One approach would be to try to calculate the particle-particle interactions but this is not computationally feasible since we would have 2^N interactions where N is the particle number and we are using thousands of particles.

2.2 Numerical methods

The simulation is set up with respect to a three dimensional grid that supports particles (Fig. 1). The overall algortihm is expressed in Algortihm 1.

Algorithm 1. Main program.

set Dirichlet or periodic boundary conditions
set initial $\mathbf{x}_p$, $\mathbf{v}_p$ and m_p for all particles p
interpolate initial ρ from particle mass m_p
set initial guess for ϕ
loop
    solve ϕ at the nodes using the Jacobi method
    calculate $\mathbf{F} = -\nabla\phi$ with respect to each particle p
    for every particle p **do**
        $\mathbf{v}_p(t + \Delta t) = (\mathbf{F}_p/m)\Delta t + \mathbf{v}_p(t)$
        $\mathbf{x}_p(t + \Delta t) = \mathbf{v}_p(t)\Delta t + \mathbf{x}_p(t)$
    end for
    interpolate new ρ from particle mass m_p
    $t \leftarrow t + \Delta t$
end loop

The Poisson's equation of gravity is discretized to

$$\begin{aligned} 4\pi G\rho_{i,j,k} = &\frac{\phi_{i+1,j,k} - 2\phi_{i,j,k} + \phi_{i-1,j,k}}{(\Delta x)^2} + \\ &\frac{\phi_{i,j+1,k} - 2\phi_{i,j,k} + \phi_{i,j-1,k}}{(\Delta y)^2} + \\ &\frac{\phi_{i,j,k+1} - 2\phi_{i,j,k} + \phi_{i,j,k-1}}{(\Delta z)^2}. \end{aligned} \tag{5}$$

The gravitational potential ϕ is then solved at each node using using the Jacobi method (Algorithm 2) . In the calculation of the potential gradient we obtain the potential values near the particles by linear interpolation and then calculate the gradient by numerical differentiation. For exmaple at the pth particle the gradient is calculated in the x direction as

$$(\nabla\phi)(x_p) = \frac{\phi(x_p + dx) - \phi(x_p - dx)}{2dx}, \tag{6}$$

where x_p is the particle's x coordinate and dx is a small distance.

The particle velocity and position are evolved over a small time step dt, assuming that the force remains constant. Essentially this means calculating

for each particle

$$= \mathbf{v}_p(t + \Delta t) = (\mathbf{F}_p/m)\Delta t + \mathbf{v}_p(t) \tag{7}$$

$$= \mathbf{x}_p(t + \Delta t) = \mathbf{v}_p(t)\Delta t + \mathbf{x}_p(t). \tag{8}$$

After this using the new positions the particle masses are linearly interpolated back to the nodes to obtain a new mass density ρ.

Algorithm 2. Calculating potential ϕ using the Jacobi algorithm.

function Jacobi(ϕ, tolerance, max_iterations)
  for $n = 0, 1, \ldots$ max_iterations **do**
    error $\sigma \leftarrow 0$
    for every node $\phi_{i,j,k}$
      $\phi_{i,j,k}^{(n+1)} \leftarrow \phi_{i,j,k}^{(n)} + (\phi_{i+1,j,k}^{(n)} + \phi_{i-1,j,k}^{(n)} +$
        $\phi_{i,j+1,k}^{(n)} + \phi_{i,j-1,k}^{(n)} + \phi_{i,j,k+1}^{(n)} + \phi_{i,j,k-1}^{(n)} +$
        $h^2\rho_{i,j,k})$
      update σ
    end for
    if $\sigma \leq$ tolerance, **break**
  end for
  return ϕ
end function

2.3 Initialization of the simultaion

The specific case we want to study with our simulation is a dark matter halo. Dark matter halos are structures composed of dark matter, believed to envelope galaxy disks.

We randomly distribute dark matter particles into a box and let them collapse by the effect of gravity. We used either Dirichlet boundary conditions with $\rho = 0$ or periodic boundary conditions.

2.4 Verification of results

To verify the simulation results we test for energy conservation:

$$\begin{aligned} T + V &= \text{constant} \\ \frac{1}{2}\sum_p m_p v_p^2 + M\sum_{i,j,k}\phi_{i,j,k} &= \text{constant} \end{aligned} \tag{9}$$

where T is the total kinetic energy, V is the total potential energy and M refers to the total mass of the particles. In the context of gravitationl interactions only, the virial theorem states:

$$\langle T \rangle_\tau = -\frac{1}{2} \langle V \rangle_\tau \tag{10}$$

where we take a time average of the kinetic and potential energy over a time period τ.

We could not use time steps, distances or masses with proper units because the Uintah interpolator had some problems with large numerical values (see section 3.1). Therefore the set of values in our simulation have arbitrary units. To test energy conservation we solved a scaling factor c from the virial theorem over some time interval τ. The scaling factor relates the kinetic energy T and potential energy V of the system via $\langle T \rangle_\tau$=c$\langle V \rangle_\tau$. We then tested whether energy conservation is satisfied when we use the scaling factor c.

We also compared the radial mass distribution of the obtained halo with the Navarro-Frenkel-White (NFW) profile, which is the radial mass distribution often obtained from simulations (Navarro et al., 1995). The profile is given by

$$\rho(r) = \frac{\rho_0}{\frac{r}{R_s}\left(1 + \frac{r}{R_s}\right)^2}, \tag{11}$$

where r is distance from the center of the halo and (ρ_0, R) are parameters.

3 IMPLEMENTATION

In this section, we will often refer to Section 2.2 and describe each numerical method's implementation in detail. During the whole project, we made a lot of effort to make sure the algorithms fit the Uintah framework. Because of this, we will have extensive description of the Uintah framework in the Section 3.1, describing how the Uintah patches which are used in the Poisson solver work, how particle masses are interpolated to the grid and how the gradient of potential is calculated.

In addition to describing our implementation and the Uintah framework, we will discuss some of the limitations and bugs of Uintah software which drained a painstaking amount of time to solve. For more information on

Uintah framework, please refer to the *uintah_presentation.pdf* we made on the software.

3.1 Uintah

As mentioned in the title, we made a *fully parallel* poisson solver utilizing particle-in-cell approach.

Before we start with the actual algorithms, let's go through the basics of Uintah. There are a few concepts to understand; *cells*, *patches*, *ghost cells*, *cell-centered variables*, *node-centered variables*, *face-centered variables* and *particles*.

We present an explanation for each of these accompanied by a Figure (Figure 1) demonstrating a visual description.

1. *Cells* are finite boxes in the grid. These are the basic elements of our simulation. In the case of our poisson solver, we discretize our poisson equation based on cells in such a way that our discretization length h in Equation 5 is the length between *cells*.

2. *Patches* are collections of *cells* in our grid. The reason to have a concept of *patches* is that Uintah uses patches to distribute workloads for different processes. In Uintah, whenever we iterate through *cells*, we iterate through them patch-by-patch. This introduces some complications, such as how to handle the boundary between patches. The boundary issue will be further elaborated on.

3. *Ghost cells* are cells which are one or more step beyond the border of a *patch*. The whole concept of ghost cells is meaningless if we do not work in parallel. In a parallel simulation (employing more than one process), having ghost cells is of utmost importance because it is used to handle *MPI communication* between different processes' *patches*.

4. *Cell-centered variables* are variables which are defined at the center of each *cell*.

5. *Node-centered variables* are variables which are defined at the corners of each *cell*.

6. *Face-centered variables* are variables which are defined at the faces of each *cell*.

7. *Particles* are point variables defined at any arbitrary point in the simulation grid. In our simulation, cell/node/face-centered variables interact with each other by interpolating particle variables to cell/node/face-centered positions and by interpolating cell/node/face-centered variables to particle positions.

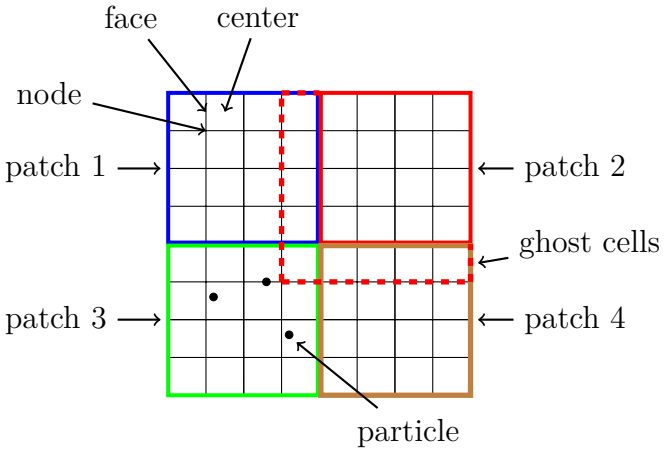

Figure 1. A two dimensional view of the grid with its various components labeled.

One of the biggest reasons we chose to use the Uintah framework was that it has a very advanced *parallel task system*. As already mentioned previously, in Uintah parallelization goes over patches. Every *process* has N patches which they are in charge of, and those patches are distributed according to load balancing settings. In our case, we chose to use a load balancing option that takes the number of particles into account.

The parallel task system takes care of most data communication and working with data dependencies. For each *function*, we must define a task which takes in the function, the function's *dependencies* and the function's *output*.

For instance, as we can see in the Equation 5, for one poisson iteration, we need the *last iteration's* $\phi = \phi_{previous}$ value as well as the distribution of particle density ρ, and one poisson iteration calculates the next $\phi = \phi_{next}$ value. Additionally, because according to Equation 5 we need neighbor values of every ϕ in our iterations, we need to tell the task that it needs to fetch 1 layer of *ghost cells*. The concept of *ghost cells* was demonstrated in Figure 1.

In pseudocode, this would be:

Algorithm 3. Demonstrating Uintah taskgraph system

$Task \leftarrow$new task($poissonIterationStep$)
$Task \leftarrow \phi_{previous}$ with 1 layer of *ghost cells*
$Task \leftarrow \rho_{current}$
$Task$ *computes* ϕ_{next}
$Scheduler \leftarrow Task$

In addition to tasks and different patch hierarchies, we may also explain the way Uintah parallelizes things, and why it is currently regarded as perhaps the most scalable grid library. Normally in most grid libraries, things advance in a *semi-serial* way due to normal MPI programming. Whenever a function is done calculating, there is some communication between MPI processes (e.g. process #1 needs to tell #2 some result of a calculation). Until the communication is done, the process is often left hanging. This phenomena is often referred to as *load imbalance*.

In the case of Uintah, the problem of load imbalance is overcome by introducing *task dependencies*, which we already went through shortly in this section. Task dependencies refer to tasks having certain dependencies before they can be executed. For instance, say process #1 can move to interpolating particle values to a grid or to calculating poisson iterations and poisson iteration still requires a variable which has not been calculated. In this case, the process #1 would simply move on to interpolating particle values, instead of hanging around doing nothing, as would be the case in most normal parallel simulations.

For demonstration, we present a task graph taken from Uintah website in Figure 2.

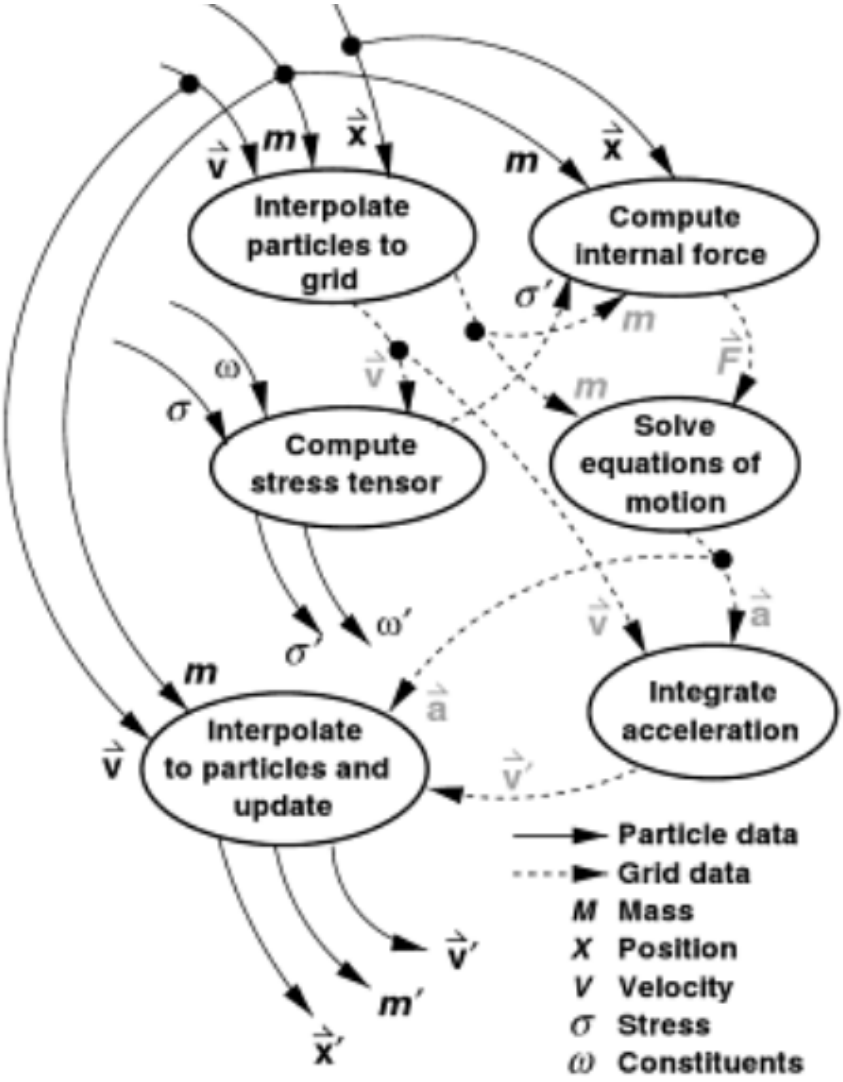

Figure 2. Task graph example from Uintah's website

In Uintah, there are essentially *2* different ways to parallelize the simulation. Either one distributes the workload among *processes*, or one distributes the workload among both *processes* and *threads*. We did not use the thread-based parallelization because unfortunately neither of our MPICH versions had support for the required level 3 parallelization, and tackling that problem simply requires too much of our valuable time. In fact, even FMI's supercomputer *Voima* did not have support for level 3 paralllelization. We were

still intrigued by the concept of making a *hybrid-parallelization*, especially since our simulation should have it working out of the box, so we read more about it.

The way hybrid-parallelization in Uintah works is that every process is assigned N threads, from which 1 is used for MPI communication, and $N-1$ are used for execution of tasks. This should guarantee fast execution on supercomputers, as long as $N \approx 15-30$. Using hybrid-parallelization would reduce MPI communication, as threads within a process would have shared memory, and would not require for instance communication of ghost cells through MPI.

In addition to having a nice task-graph system, Uintah has put some effort into having general-use interfaces for implementing things. Patches can be sorted with *Hilbert spatial filling curves*[1], Uintah has tools for interpolating particles to nodes and nodes to particles and it has tools for calculating interpolated derivatives. Additionally, Uintah has support for implementing *adaptive mesh refinement*[2]. Despite the fact that these tools have a steep learning curve, they will become invaluable when implementing parallel and optimized algorithms for mesh-based methods.

Finally, we will discuss some of the *bugs and limitations* in Uintah. For one, interpolating particle and cell variables back and forth are viable only for node-based variables out-of-the-box. This is not unusual, as the visualization software *VisIt* and most other mesh visualization softwares have a similar take on this. Making sparse data is also not possible in Uintah, meaning that solving e.g. *Vlasov* equation on a sparse grid is not possible, and neither is solving many-dimensional partial differential equations with methods such as adaptive *radial basis method*[3]. Additionally, there was one great limitation we encountered; the particle interpolation does not work when the particle variable has values around $1.0e40$. It gives spiked results for the interpolated variable, and it took us a day to find out this was why our simulation was producing incorrect results.

[1]Hilbert spatial filling curves are used to optimize memory cache. In the case of our simulation, cells are sorted in the way that neighbors of each cell are closer in memory, reducing cache misses in fetching neighbor cell data. For more information on space filling curves, see `http://www.diss.fu-berlin.de/diss/servlets/MCRFileNodeServlet/FUDISS_derivate_000000003494/2_kap2.pdf?hosts=`

[2]See Section 5

[3]Solving partial differential equations by assuming a basis of radial functions and often neglecting the functions which contribute less

For some additional information, please refer to the Uintah presentation we made which can be found in:

`./doc/Uintah/uintah_presentation.pdf`

3.2 The program

Now that we know the basics on Uintah, we can move on to looking at our program. Our main algorithm is described in the Algorithm 1, and here we attempt to break this down into smaller details. All of this can be found also in our project files *ParticleTest1.cc* and *ParticleTest1.h*.

set Dirichlet or periodic boundary conditions. Setting either dirichlet or periodic boundary conditions is fairly straightforward. In fact, this is something that we found Uintah can do for us. We need to simply input our boundary conditions in a run file and take it into account in our poisson solver. In our poisson solver, we simply skip iterating the cells that are on our grid border.

set initial $\mathbf{x}_p$, $\mathbf{v}_p$ *and* m_p *for all particles* p. For this purpose, Uintah specifies a function for initializing our data. We set particle position $\mathbf{x}_p = rand(grid)$, velocity $\mathbf{v}_p = 0$ and mass $m_p = 1.0e4$.

interpolate initial ρ *from particle mass* m_p. For this, we use linear interpolation method. Our ρ is specified at each cell corner ($node$[4]) and we interpolate the particle mass variable to nearest 8 nodes. Since ρ is density, we also divide the interpolated mass variable by one cell volume $\Delta x \Delta y \Delta z$. The reason we chose to interpolate only to 8 nearest nodes (e.g. use linear interpolation) is because it can be shown with some effort that this is produces a sufficient accuracy for the Poisson equation. By sufficient, we of course mean that it does not introduce a global error of higher order than what is already in our simulation due to discretizing poisson equation. For this purpose, we found that Uintah has a class called *ParticleInterpolator* that produces interpolation. This required us to change our ρ variable to node-centered form from cell-centered form, however. This method is widely used for communication between grids and particles, and commonly methods using this are referred to as a *cloud-in-cell* or *particle-in-cell* methods.

set initial guess for ϕ. We set this to zero everywhere.

solve ϕ *at the nodes*. For this, we created an iterative solver that goes through each patch[5] and iterates through each *node* in that patch. Basically,

[4]See section 3.1

[5]See Section 3.1

we solve the Equation 5. We require that each patch fetches 1 layer of *ghost nodes*[6] to deal with patch boundaries, as our algorithm needed the neighboring nodes[7] for each node. Additionally, we skip the cells at the patch *boundaries* due to boundary conditions. Our algorithm for solving the poisson equation is demonstrated in Algorithm 2. For every iteration, we also created a task to utilize the full functionality of *Uintah*. For task dependencies in the Poisson solver, please refer to Algorithm 3.

calculate $\mathbf{F} = -\nabla\phi$ *for each particle* p. For this, we use linear interpolation. For every particle p, we pick the 8 nearest nodes and interpolate to values $x_i + dx_i$ and $x_i - dx_i$ for every i. This means that our gradient calculation is also based on linear interpolation. This introduces some error, but again it has been shown to be sufficient for solving poisson equation.

propagating every particle p. This is already documented in Algorithm 1, so we do not elaborate it further here.

interpolate new ρ *from particle mass* m_p. See the step *interpolate initial* ρ *from particle mass* m_p.

$t \leftarrow t + \Delta t$. This is done in the Uintah framework and is determined by the variable *Delt* in the runtime *XML* file. We use constant timestepping, so this step needs nothing complicated.

4 RESULTS

4.1 Movies

We present movies separately from this report. They are one of the main results of this whole study, however, as they demonstrate perhaps the best way how our simulation works dynamically. These movies can be found in the *movies* folder.

4.2 Dark matter halo

Here we present the dark matter halo in our simulation at an Equilibrium. For a better view of the dynamics of the whole process, please see our movies of different simulation runs.

[6] See Section 3.1

[7] neighboring node = node which is one node to the left,right,down,up,forward or backward

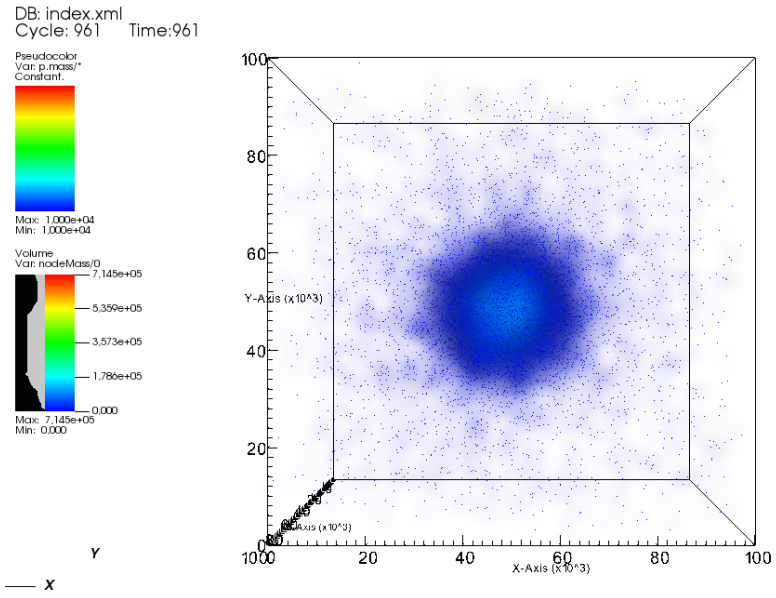

Figure 3. Our simulation run with periodic boundary conditions and a formed halo. The simulation time step is chosen from a point when the halo shape is at an equilibrium.

4.3 Energy conservation and virial theorem

We want to make sure that energy is conserved in our simulation and we compare it with the virial theorem. Energy conservation is somewhat difficult to do because of the weird units we were forced to use which were Section 2 (namely $G = 1$).

For simplicity, we calculate a *scaling factor* C for comparing units of ϕ and $|\mathbf{v}|^2$ by using the virial theorem:

$< v^2 >_\tau = C < \phi >_\tau$

In other words, $C * \phi$ is simply ϕ in same magnitudes as v^2. As for why we need to do this is because Uintah simply does not have support for interpolating large values back and forth, as mentioned in Section 3.1.

We then compare energy conservation by comparing different timesteps' total energy:

$[0.5v^2 + C\phi]_{t=t_1} - [0.5v^2 + C\phi]_{t=t_2} = Error$

In this case, we only look at 10 timesteps.

The results are presented here and under
./doc/results/energyConservation/virialTheoryAndEnergyConservationResults.dat:

```
Scaling factor, calculated from virial theorem: -2.00313776377e-05
Energy conservation at timestep 140; potential: 2171895072.08, v_squared: 21895902
Total Energy: 4361485334.96
Energy conservation at timestep 141; potential: 2171448825.2, v_squared: 217791263
Total Energy: 4349361457.24
Energy conservation at timestep 142; potential: 2171267304.94, v_squared: 21488164
Total Energy: 4320083745.2
Energy conservation at timestep 143; potential: 2170731390.9, v_squared: 21886569
Total Energy: 4359388376.3
Energy conservation at timestep 144; potential: 2169414726.35, v_squared: 21723109
Total Energy: 4341725646.81
Energy conservation at timestep 145; potential: 2167363889.69, v_squared: 21508854
Total Energy: 4318249331.79
Energy conservation at timestep 146; potential: 2165176115.51, v_squared: 21526198
Total Energy: 4317795976.84
Energy conservation at timestep 147; potential: 2163085388.64, v_squared: 21725016
Total Energy: 4335586989.23
Energy conservation at timestep 148; potential: 2161168195.91, v_squared: 21783086
Total Energy: 4339476877.23
Energy conservation at timestep 149; potential: 2159870013.0, v_squared: 21398180
Total Energy: 4299688108.84
```

It looks as though our energy is not conserved perfectly, but it is in the right orders of magnitude. The largest change (between first and last timestep) for example is merely $E_{diff} = 61797226$, whose relative value is $61797226/4299688108 \approx 0.014$. We assume this depends on both the choice of *dt* and *resolution* in our algorithm. This will be discussed in the Section 5.

4.4 NFW profile

As mentioned in Section 2.4, the dark matter halo density profiles should follow the *NFW* profile given in Equation 11. This is when a dark matter halo has reached an equilibrium, so at a timestep quite far away in the simulation. We made multiple runs, of which we will use the run which used 50^3 cells

and 150000 particles and periodic boundary conditions. We take a line-out plot of the dark matter density ρ and compare it with the *NFW* profile.

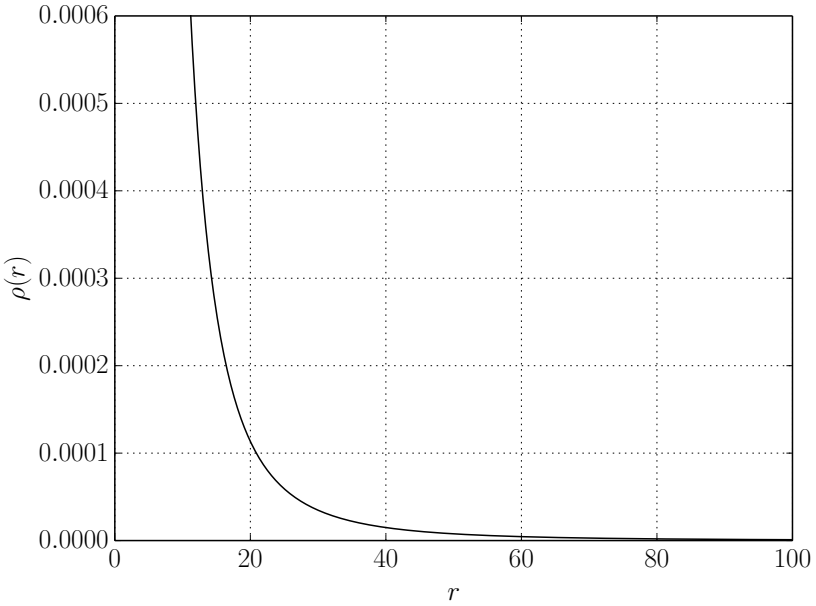

Figure 4. The shape of the radial distribution of mass density suggested by NFW profile (Equation 11), choosing $\rho_0 = 1$ and $R_s = 1$.

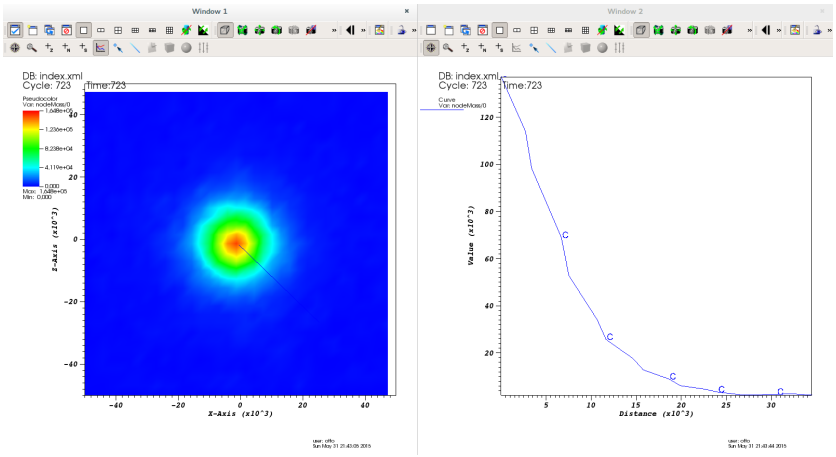

Figure 5. Line-out from our simulation. The line-out starts from the center of the dark matter halo

From comparing at Figures 4 and 5 we simply conclude that the NFW profile could be fitted into the profile of the dark matter halo. Our peak is not as strong as we have a discretized grid which prevents our values from hitting ∞. Obviously, the Figure 4 is only a plot of the *NFW* profile with coefficients we picked by ourselves. We show it mainly to give the reader a general view of the shape of the *NFW* profile.

Our initial plan was to actually fit the Equation 11 into our simulation, but unfortunately *VisIt* visualization tool could not allow us to extract variables from a lineout plot so we could do it in Python. Hence, we instead compared to make sure the *NFW* profile's shape correlated with our lineout plot

4.5 Scalability

With the boasted great scalability in Uintah, we decided to run our code on the supercomputer *Voima*. For parallelization, we used an MPI-based approach, so we use processes for parallelizing the threads and use 1 thread per process. This is due to lack of support for level 3 parallelization in Voima's *MPICH* library. According to various sources, this could be fixed either by compiling MPICH again from source with different configuration settings or possibly by tampering with some flags. Anyway, we made scalability tests with 10,20,30,40,50,60 and 70 nodes by running a simple test case with 5 million particles, 300^3 cells and 5^3 patches. The log file of the runs is presented in the *Uintah_run_information* folder of the root folder.

For results, we present Figures 6, 7.

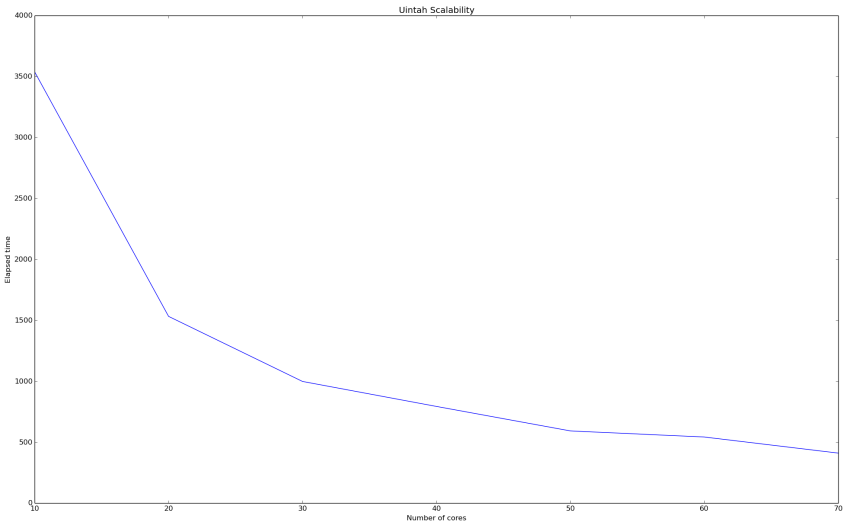

Figure 6. Scalability test for our simulation. We ran the simulation with 5 million particle and 300^3 cells.

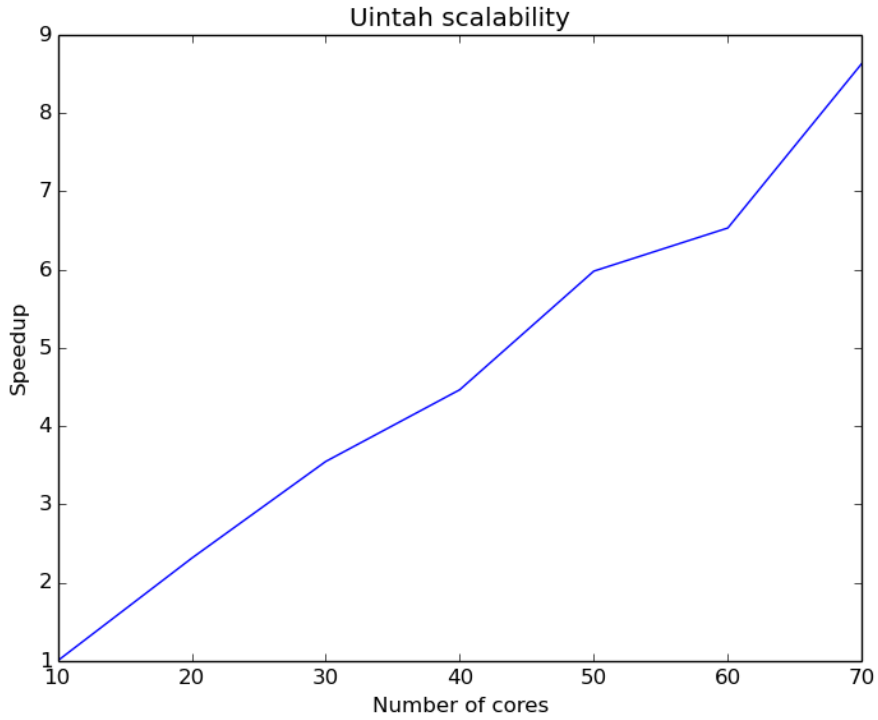

Figure 7. Scalability test for our simulation. We ran the simulation with 5 million particle and 300^3 cells. The speed-up is counted relative to 10 cores

Most notably, we see a speed-up of over 8 in Figure 7. This can only be explained by some memory optimizations in different processes running on different cores. For example, it is known that with large datasets and with large memory usage, *cache misses* become more frequent as large jumps in memory become more frequent.

We would have also tested for *hybrid parallelization* by employing both processes and threads, but unfortunately that would have required tampering with the MPICH library, which would have cost quite a bit of time.

5 CONCLUSIONS

This project had two main purposes. Firstly we simulated a dark matter halo by solving the Poisson's equation of gravity coupled with dark mat-

ter particles in the framework provided by Uintah. Secondly, and perhaps more importantly, we learned to use some of the powerful tools provided by Uintah that provide parallelized and optimized grid framework for solving a multitude of problems in computational physics.

The simulation results are physically sound as they comply with energy conservation and virial theorem and reproduce a mass density distribution similar to other simulations.

One important Uintah tool that we did not utilize due to time constratints is adaptive mesh refinement (AMR). This essentially means that based on some predefined condition(s) specified in the simulation, the resolution would be increased in some parts of the grid. This would let us emphasize interesting regions in the simulation and de-emphasize less interesting regions. This also means that the simulations becomes computationally more efficient. AMR is something we wish to explore more in the future.

NFW profile could be fitted to the data from our simulation to find out the parameters (ρ_0, R_s). These parameters could then be compared with other studies. We also want to make the simulation with proper units and realistic scale, this would probably mean solving some problems with the Uintah interpolator. Also the the effect of resolution on the accuracy of the results could be studied more extensively, both analytically and numerically.

6 RUNNING, COMPILING AND DOING THINGS WITH UINTAH

In the requirements for the course, it was encouraged to add the software used into the project folder. We will not do this, as Uintah is a large code that has some $\approx 700k$ lines of code. Additionally, to analyze the files one needs to compile *VisIt* from source with the –uintah option in order to read the *.uda* file format.

In other words, installing Uintah can be done by first downloading it, taking a 2-10 hours to install all dependencies needed to compile it, then compile it (the compilation will take another hour) and then you need to install VisIt from source. By the time you have Uintah installed, you should have most of the dependencies worked out, though, so it's a bit easier. Downloading all the stuff that *VisIt* uses will take approx an hour as well, though, and compiling VisIt takes about an hour.

For us, this took about 2 days, but then again we had some major dependency issues.

6.1 Installing instructions

Step 1 Download Uintah
Step 2 Compile Uintah in a build folder
Step 3 Download VisIt
Step 4 Install VisIt from source using the –uintah option
Step 5 You're done

REFERENCES

Kuhlen, M. et al., (2012), Numerical Simulations of the Dark Universe: State of the Art and the Next Decade, arXiv:1209.5745 [astro-ph, physics:hep-ph]

Roos, M., (2010), Dark Matter: The evidence from astronomy, astrophysics and cosmology, arXiv:1001.0316 [astro-ph]

Hockney, R.W. and Eastwood, J.W., (1985), Computer Simulation Using Particles, CRC Press

Navarro, F.N. et al., (1995), The Structure of Cold Dark Matter Halos, arXiv:astro-ph/9508025