SIMULATING A SPHERICAL DUST CLOUD COLLAPSE IN ENZO-E

Joshua Smith | MURPA Research Project

Supervisor: Dr. Michael Norman

Contents

Aim

This project has the scientific goal of simulating the initial gravitational collapse of a molecular cloud, modelled as a uniform spherical cloud comprised of particles. The technical goal is to evaluate the performance of two different gravity solvers, BiCGSTAB and Domain Decomposed, run in Enzo-E with adaptive mesh refinement (AMR). Comparisons will be made for collapse simulations with varying block sizes and processor counts.

This report summarises the results from the different simulations and contains suggestions for updates to the Enzo-E documentation, some persistent errors, and links to the scripts for use continuing the project.

Introduction

Molecular Cloud Collapse

Star formation begins when large (5+ parsec diameter), low density (~10^{-21} to 10^{-20} g/cm^3) molecular clouds consisting of mostly molecular hydrogen, collapse under their own gravity. This collapse continues at a relatively constant 10K temperature until the core reaches densities of around 10^{-13}g/cm^3 at which point the core becomes optically thick. This causes gas temperature to increase with density, enabling a pressure gradient that sustains the first hydrostatic core. [1] The hydrostatic core collects material, heating up and becoming denser until it reaches ~2000K and 10^{-8} g/m3, when H_2 dissociates and isothermal collapse occurs again. It is at this point there is a potential to form close binary stars via fragmentation. Once dissociation is complete, the collapse halts once more and the second hydrostatic core forms. [2]

Producing a simulation that is able to follow the whole collapse, from the full molecular cloud down to the second hydrostatic core, would provide a possible insight and verification of the current binary star forming theories. The difficulty is that starting with the full size means resolving details in length scales ranging 7 orders of magnitude and densities ranging from 10^{-18} to 10^{-2} g/cm^3 [2]. This makes static mesh simulations unfeasible so Adaptive Mesh Refinement (AMR) is required. An 'enclosed mass' mesh refinement criteria, where grid blocks refine if the enclosed mass exceeds a given value, is best suited because there is a well-defined minimum Jean's Mass for the collapse that provides a limit to the resolution required. [2]

This project will focus on simulating the initial stage of the cloud collapse, which is isothermal and pressure can be considered negligible.

Enzo-E and Adaptive Mesh Refinement

As the scale of hydrodynamic and N-body simulations continues to grow in the field of Computational Astrophysics, with dark matter particle simulations recently crossing the 1 trillion-particle threshold [3], new adaptive mesh refinement (AMR) frameworks are required to efficiently handle the simulations. This section will introduce the software used for this project at the San Diego Supercomputer Center (SDSC), UCSD.

Cello

Cello is an extreme-scale AMR software framework under development at SDSC. Cello implements array-of-octree AMR. This means that a spatial problem is divided into a 3D root grid of 'blocks' that are stored in an array. If a specified refinement criterion is met by a certain block, it splits in half along each spatial direction, generating 8 sub-blocks. The sub-blocks are stored in an octree data structure as descendants of the parent root block. Every leaf block is comprised of an NxNxN mesh of elements at which the model parameters are stored and evaluated. Figure 1 shows an example of a 2D mesh undergoing two layers of refinement from a 4x4 root grid.

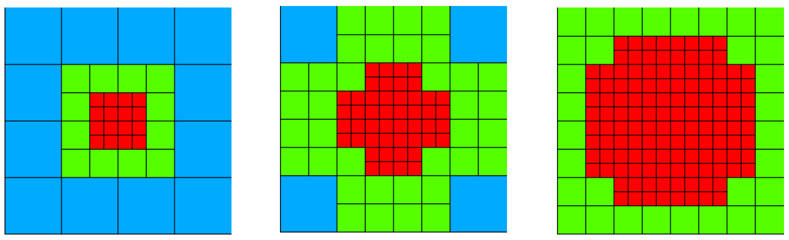


Figure 1: Mesh Refinement in 2D simulation

Enzo-E

Enzo-E is a hybrid Eulerian-Lagrangian astrophysics and cosmology application [4] being developed concurrently with Cello. It implements hydrodynamics, particle mesh self-gravity and cosmological expansion functionality inherited from its predecessor ENZO, with the Cello framework allowing for greater scalability.

The gravitational collapse analysed in this project runs by evaluating a global Laplace equation of gravitational potential at each of the mesh nodes. This is natural for field values present at each node, but particles have independent positions and velocities. A Particle Mesh (PM) method is used which maps the gravitational influence of the particles onto nearby nodes, and then the gravity equation is solved at the nodes. The solution is then interpolated from the nodes to determine the motions of the particles for that time-step.

Charm++

Enzo-E and Cello achieve parallelization through Charm++, a parallel objects system developed at UIUC. The basic premise is that the problem is over-decomposed into chunks called 'chares', which are distributed to the different processors, with some structured interaction between them (for example to request values at adjacent nodes). In this project, the chares are the leaf nodes, and there are generally many chares per processor.

Different Gravity Solvers

Two different gravity solvers will be compared in this project. The Biconjugate Gradient Stabilised Method (BiCGSTAB) is an iterative method that solves the global system of linear equations, converging to a specified tolerance over its iterations. The Domain Decomposed (DD) method is a composite method which combines a global solution over the root grid, with local tree solutions calculated at each of the root blocks. It works by projecting the global solution to be the boundary conditions of each of the local tree solves, combining the tree solves and applying Jacobi smoothing. DD is an approximate method and introduces systematic errors, but it is significantly faster than BiCGSTAB for a given problem size.

Results

BiCGSTAB Scaling

A collapse test using the biconjugate gradient stabilized (BiCGSTAB) solver was run in Enzo-E as a base case for comparison with the Domain Decomposed (DD) solvers. Key areas investigated were the scaling of time-to-solution with differing problem sizes, implemented by changing the maximum refinement level for the AMR grid, and the optimum number of processors for a given collapse size.

Performance scaling for differing maximum refinement levels

In this test, a particle only 3D collapse simulation with 4^3 root blocks was run on a single Comet node using all 24 cores. Each block has 16^3 regions. An 'enclosed particle mass' mesh refinement criteria was applied to refine the grid over the regions particles occupy. This resulted in an initially refined, static mesh throughout the collapse. The maximum mesh refinement level (level max) was varied from 2 to 4. The simulation was intended to run for 100 cycles, but the level max 3 and 4 cases were stopped early as the collapse occurred within a shorter timeframe.

A summary of the problem size for each case is shown in Table 1. Note that the particle count is a function of the initial number of grid blocks, and since particles have constant mass, increasing the maximum refinement level reduces the free fall time of the collapse.

Table 1: Effects of varying maximum refinement level

Maximum Level	Particle count	Number of leaf blocks	Simulation time at collapse (s)
2	1088	176	86
3	8744	1240	29.67
4	70200	3032	10.5

Figure 2 shows the wall time taken per cycle of the simulation. The level max = 3 case takes over ten times as long as the level max = 2, which is to be expected as it deals with 8 times the particles and 7 times the number of blocks in the mesh.

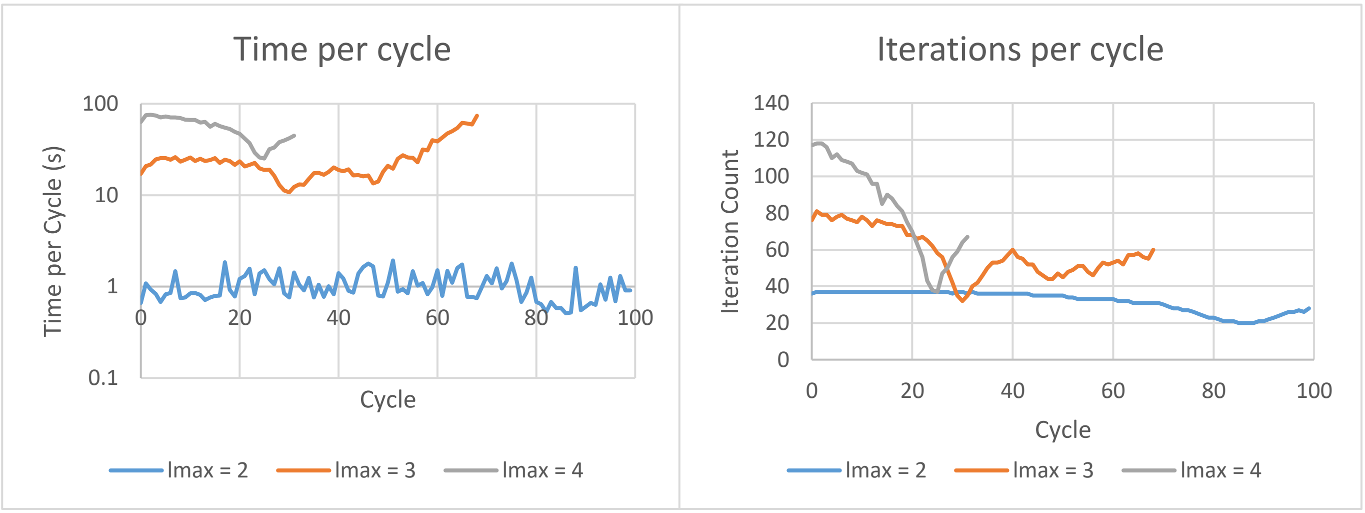
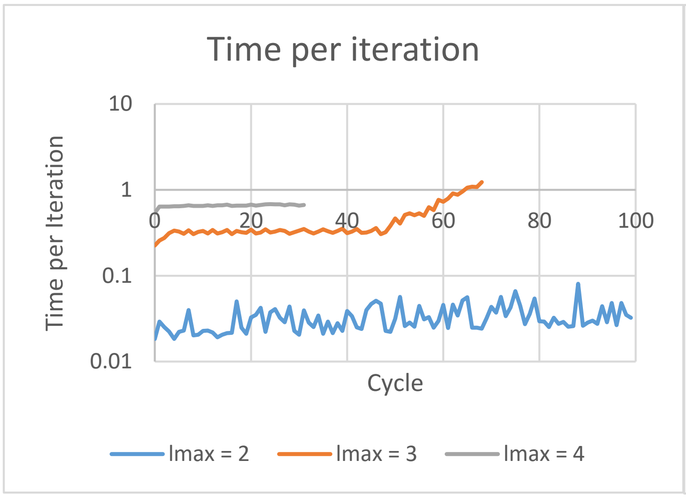


Figure 2: BiCGSTAB cycle time and number of iterations for different refinement levels

The time per cycle for each collapse can be broken into the time per iteration of the BiCGSTAB solver, and the number of iterations required to achieve the specified tolerance. The number of iterations per cycle appears inversely related to the spread of the particles. The number of iterations drops as the particles collapse and rises as the particles spread post collapse. The local minima in Figure 2 pinpoint the time at singularity for each simulation.

Figure 3 shows the time per iteration. It is relatively constant for each method indicating that it does not have the same dependence on particle position/velocity as 'iteration count' does. The upward slope at the end of the *level max = 3* test is due to additional mesh refinement as particles spray outwards following the collapse.



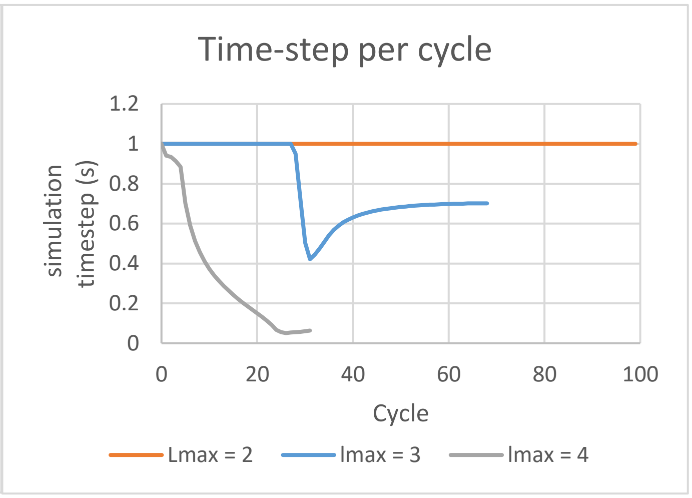


Figure 3: BiCGSTAB iteration time and simulation time-step for different refinement levels.

The time-step per cycle as shown in Figure 3, is determined so that no particle will travel more than one grid block. This is why the time-steps for the refined cases decrease as the cloud collapses and the particles are moving faster. A maximum time-step of 1 second was implemented because initially the particles are at rest, resulting in an otherwise unbounded time-step.

Performance Impact of additional processing elements.

The collapse test used in the max level = 2 case from above was run again, but this time varying the number of cores used from 4 to 24.

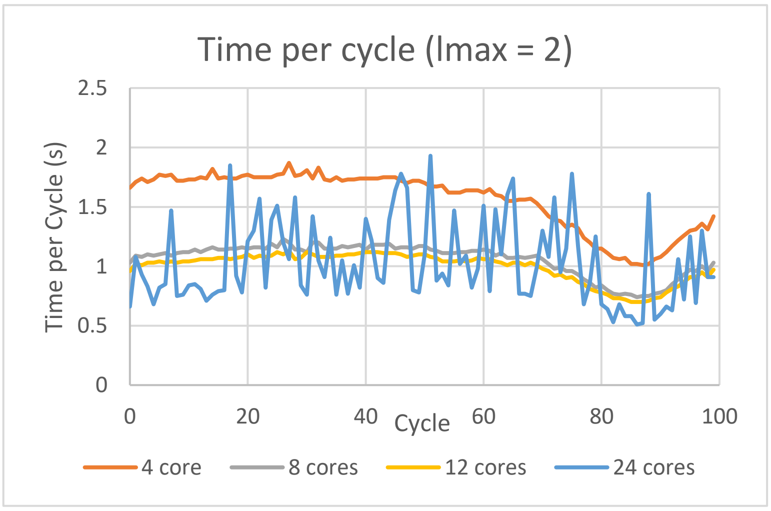


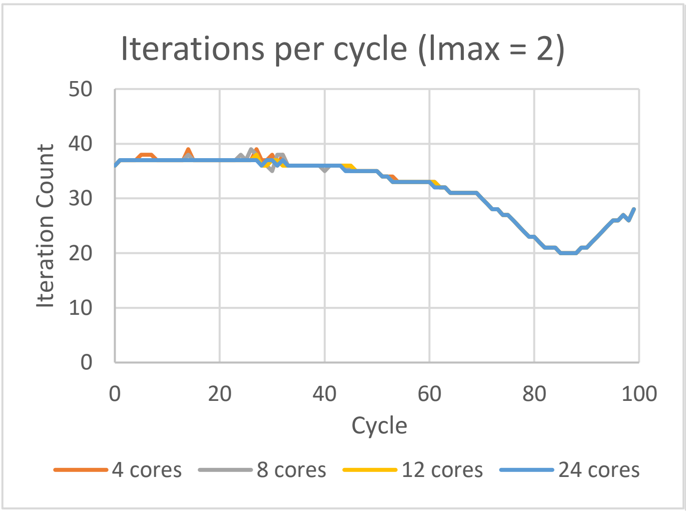
Figure 4: Influence of number of processors one time to solution of BiCGSTAB collapse

Figure 4 shows that increasing from 4 cores to 8 yields a collapse in 67% of the previously required wall time. This is longer than the theoretical limit of twice the speed due to doubling the computing power. Increasing to 12 cores provides a minimal speed increase suggesting there is a sweet spot the ratio of problem size to number of processor elements (PE) used. The 24 core test shows highly variable time per cycle, sometimes taking much less or more time than the other tests, this could indicate uneven distribution of tasks between processors, where some processors sit idle and others have an excess of jobs to complete. Table 2 summarizes the average time taken for each test:

Table 2: Influence of number of cores on speed of level max =2 BiCGSTAB collapse

Cores used	Blocks per PE	Average wall time per cycle
4	44	1.56
8	22	1.05
12	14.67	0.99
24	7.33	1.02

Figure 5 shows that the cycle time differences come from the time taken per iteration, as the iteration count per cycle is equal in each collapse. The equal iteration count is to be expected as the same physical problem is being run on identical solvers, only varying the degree of parallelism.



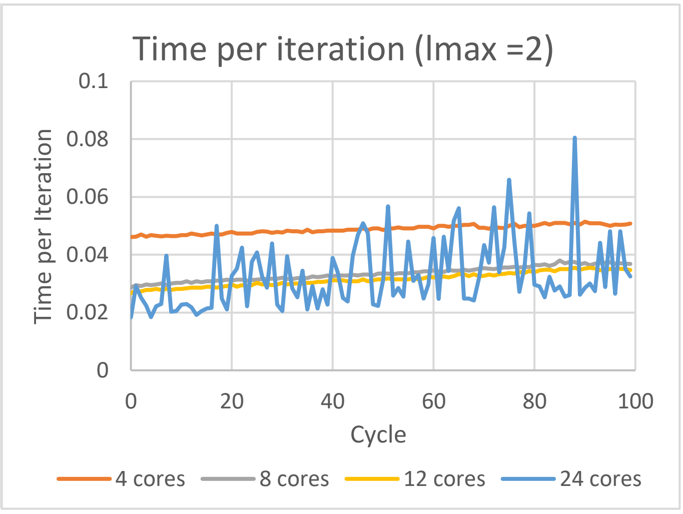


Figure 5: Breaking the cycle time into 'time per iterations' and 'number of iterations' for level max = 2 BiCGSTAB collapse.

BiCGSTAB and DD comparison

Regular collapse with large DD errors

As the Domain Decomposition (DD) solver provides an approximate solution, in comparing the two methods both solution accuracy and time-to-solution were analysed.

Initially a 4*4*4 root grid collapse similar to the level max = 3 case from the BiCGSTAB section above was run on both solvers. It became apparent that the DD method was producing significant visible errors as shown in Figure 6.

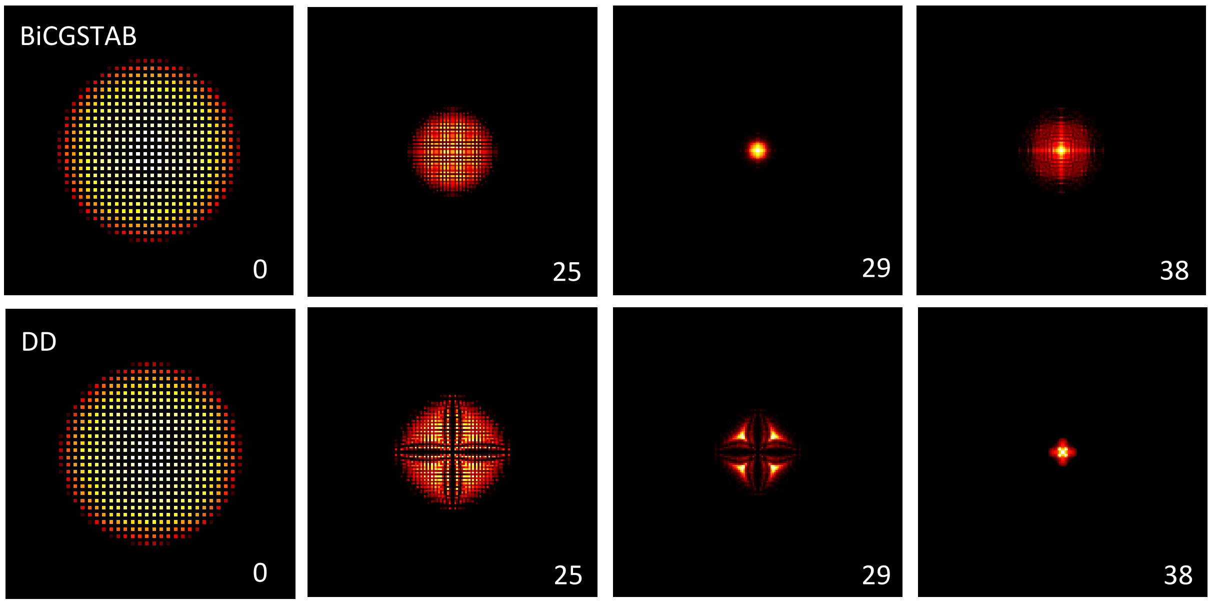


Figure 6: Particle Positions for BiCGSTAB and DD solutions throughout collapse, listed by cycle

The fact that the DD simulation appears to collapse to four (8 in 3D) points appears to be due to underestimating the gravitational influence from adjacent root blocks. The DD solver generates the solution to the gravity equation at the scale of each root grid individually, and then stitches that 4*4*4 grid together and applies smoothing to achieve its approximate solution. Having the collapse centred exacerbates the errors as it surrounds a vertex spanning 8 different root grid blocks. The DD collapse also takes longer than the BCG base case, with the first singularity occuring well after the base case. (cycle 29 vs cycle 38).

On the other hand, the DD method is far faster at computing a problem of this size. Figure 7 shows that DD can be up to 8 times faster at simulating a cycle of the collapse.

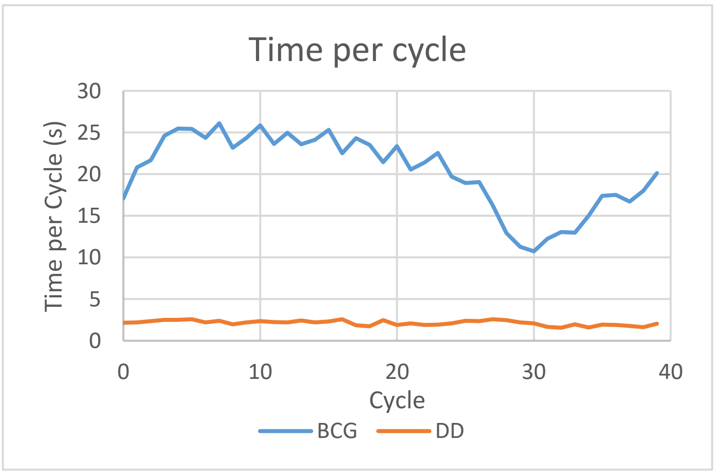


Figure 7: Wall time comparison between BiCGSTAB and DD for collapse

Offset collapse to the centre of root grid block

A natural progression is to try the collapse when centred on a root grid block instead, as there will be less interaction between particles from adjacent root blocks, and therefore potentially less error. This was achieved by offsetting the spherical mask that determines where particles are placed in the solution space.

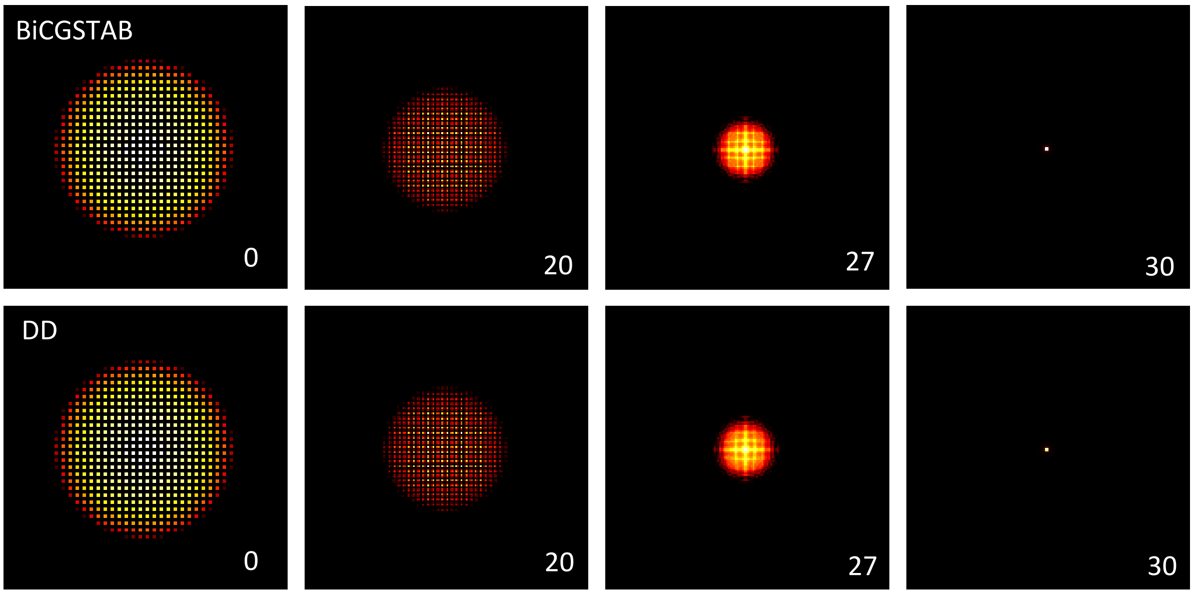


Figure 8: Particle Positions for BiCGSTAB and DD solutions throughout improved offset collapse, listed by cycle

Figure 8 confirms this produced much more accurate results, with both collapses visually indistinguishable and singularity occurring simultaneously. However, the load unbalance caused from all of the particles being within a single root grid block negatively affected the performance of both methods. This caused the BCG solver to take an extra 3 or 4 seconds per cycle and had a much larger impact on the DD solver, which now takes over ten seconds per cycle compared to the 2-3 seconds required in the original case.

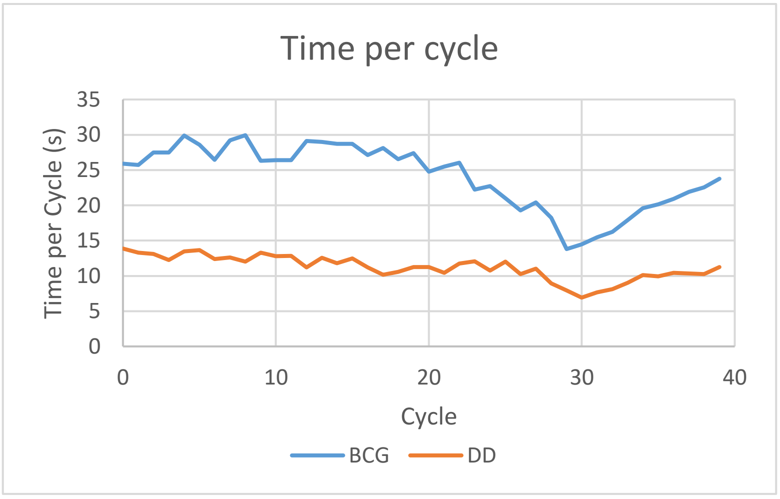


Figure 9: Wall time comparison for the offset collapse simulation.

Comparing accuracy of DD results

In order to quantitatively compare the accuracy of DD with respect to the BCG solution, the L2 error norm was used. This is effectively the Euclidean distance of all the error residuals in the 3D data cube divided by the Euclidean distance of all the true values:

$$L2\ error\ norm = \frac{\sqrt{\sum(x_{approx} - x_{true})^2}}{\sqrt{\sum x_{true}{}^2}}$$

I wrote a Python script that scans through the files produced by each processor at each timestep, and collects the residual errors for a sample of data points in each file, from which the L2 error norm is calculated.

Originally the gravitational potential field was the obvious field for comparison as it is the primary solution to each simulation step, from which others are derived. However, the potential fields from each method were vastly different (around -4e29 for BCG vs +2e29 for DD) so the acceleration in the x direction field was compared instead.

The first case produced an overall L2 error norm of 0.2999. The offset case had an improved L2 error norm of 0.1536. The DD solver in each case was set up to run with a resolution tolerance of 5% in its coarsest solving stage.

Conclusion

For a spherical cloud collapse, BiCGSTAB scales to take roughly 10 times as long for each extra level of refinement. Note that this is for an initially refined but static grid, due to issues with the particle set-up routine. The test varying the number of processors for a given problem indicates that 22 leaf blocks/chares per processor yields close to optimum performance, with adding more processors giving diminishing returns.

The Domain Decomposed method offers far better (up to 8 times faster) performance than BiCGSTAB, but produces large errors at the connecting faces between the root grids which makes its overall accuracy highly dependent on the problem layout. Future areas of research include finding whether the DD errors observed were due to bugs in the implementation, or are characteristic of the method itself. It would also be interesting to compare the two solvers accuracy given problems that require equal wall-time, as DD can refine further to reduce error, due it its increased speed.

Suggestions for Enzo-E documentation updates

Here I list some of the areas where the tutorials hosted at http://client64-249.sdsc.edu/cello/ have become outdated or things that may be helpful to include.

Parameter Files

- In the 'Subgroups' section, under naming shorthand, the order of 'Initial : density : value' should be 'Initial : value : density' to match the example.

Running Enzo-E on Comet:

- CELLO_ARCH, needs to be set to comet_gnu now, not gordon_gnu

```
export CELLO_ARCH=comet_gnu
```

- Add charm to path for ease when calling 'charmrun'.

```
export PATH=$PATH:~/charm-6.9.0/bin
```

- After installing Charm++, update the CHARM_HOME variable to tell the configuration file where it is installed on Comet.

```
export CHARM_HOME=$HOME/charm-6.9.0
```

- HDF5 is present on Comet as a module so linking to James' HDF5 directory it is potentially not required.

```
module load hdf5-serial
or
module load hdf5
```

- If you are going to be running many Enzo-E applications, it can be useful to make a shell script that purges and loads the correct modules (And possibly call it from .bashrc, to prevent your bash scripts failing each time you forget):

```
#ensures module commands inherited
source /etc/profile.d/modules.sh

module purge
module load python
module load boost
module load gnu
module load hdf5-serial

echo "Enzo-E modules loaded!"
```

  Then you can source this shell script from .bashrc to load the modules upon login to comet.

- Prior to units being implemented, CELLO_PREC should be set to double to avoid overflow errors.
- If you get the following error upon building Enzo-E:

```
warning: Ignoring missing SConscript '/home/user/public/Grackle/src/clib/SConscript'
```

  It is because Grackle is not installed on Comet. If you do not need to use Grackle, you can edit the top-level SConstruct file and set "use_grackle = 0", then Enzo-e should install correctly without it.

- Running Enzo-E no longer requires the '++runscript ./run-enoe.sh' mentioned in the tutorial, instead it requires '++mpiexec'.

```
charmrun +p8 ++mpiexec bin/enzo-p input/method_ppm-8.in
```

- Upon running Enzo-E on an interactive or compute Comet node, if you receive the following error complaining about a Nvidia library file:

```
/home/user/enzo-e.jobordner/bin/enzo-p: error while loading shared libraries:
libXNVCtrl.so.0: cannot open shared object file: No such file or directory
```

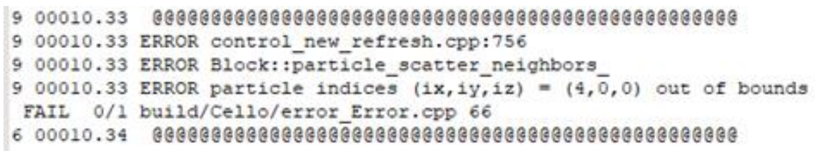
  It occurs because the libXNVCtrl.so.0 file is present in /usr/lib64 in the login nodes but not the compute nodes. One workaround is to recompile Charm++ and Enzo-E on a compute node, or you can copy the missing file from the login node to your home directory and add that directory to LD_LIBRARY_PATH as follows:

```
cd ~
mkdir lib
cd lib
cp /usr/lib64/libXNVCtrl.so* .
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/lib
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:\$HOME/lib" >> ~/.bashrc
```

Current Enzo-E Issues

1. Particles are implemented with a fixed mass, rather than by density. And the number of particles scales with the initial root refinement level.
   This means that different maximum refinement levels produce collapses with a different free-fall time. It also causes issues with refining the grid based on enclosed mass, since the initial grid refining one step creates more particles which creates more mass causing further ’runaway’ refinement. For this reason, I was only able to achieve ‘all or nothing’ refinement for the initial particle sphere.

2. If I run the working 2D collapse with a max_level of 5 or more, it crashes with the following error in the second iteration.



```
9 00010.33  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
9 00010.33 ERROR control_new_refresh.cpp:756
9 00010.33 ERROR Block::particle_scatter_neighbors_
9 00010.33 ERROR particle indices (ix,iy,iz) = (4,0,0) out of bounds
 FAIL  0/1 build/Cello/error_Error.cpp 66
6 00010.34  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Links to Relevant Scripts and Animations

The external files associated with this project are hosted at the following GitHub repository:

https://github.com/joshesmith/Enzo-E-outputs-and-helper-scripts

Note that the scripts which interact with Enzo-E outputs may need to be updated as Enzo-E develops.

Collapse Animations

The subdirectory *Collapse_Simulation_gifs* contains animations of the particle matter as it undergoes collapse, sorted by solver used and maximum mesh refinement level.

Generating Timing Spreadsheet from Enzo-E Standard Output

The subdirectory *create_timing_spreadsheets* contains two Python files that scan the standard output of an Enzo-E collapse, extracting timing and block information and exporting it to a csv for use in graphs and figures.

They are configured to run from the Linux or Windows terminal, where the first argument is the Enzo-E output file, and the second argument is the destination CSV.

A set of default inputs and outputs is present in the directory to illustrate usage.

Calculating L2 Error Norm between Simulations

The subdirectory *calculate_L2_error_norm* contains two terminal Python scripts, both which compare pairs of HDF5 files and calculate the L2 Error norm based on the first file as the true case.

Compare_errors_two_files.py, takes two input HDF5 file names (or paths if not in current directory), one field name and one destination text file name.

The input files need to be the exact same block and time-step but from a separate simulation. This means that if the mesh has refined differently in each simulation, it is unlikely a comparison can be successfully made. Static meshes work best for this reason.

Compare_errors_two_folders.py, works the same as above, except it takes two directories, not files, as inputs. The directories should contain a collection of HDF5 files for the same blocks and time-steps between two simulations. This script iterates through the files, and calls the *compare_errors_two_files.py* script on each pair, before finally combining the errors into a global L2 error.

References

[1] D. Price, “Star Formation,” *ASP2062 Introduction to Astrophysics*. Monash University, 2018.

[2] M. R. Bate, “Collapse of a Molecular Cloud Core to Stellar Densities: The First Three-Dimensional Calculations,” 1998.

[3] M. L. Normal, “Asynchronous Task-Based AMR with Distributed Parallel Objects,” in *SIAM PP20*, 2020.

[4] J. Bordner, “The Enzo-E/Cello Project.” [Online]. Available: http://cello-project.org. [Accessed: 20-Feb-2020].