PaINS: Parallel Inviscid Navier-Stokes solver

Peter Harrington
Scientific Computing & Applied Mathematics
UC Santa Cruz

June 12, 2018

Contents

1 Introduction

This project has evolved slightly throughout the quarter, and due to the finite number of hours per day, days per week, and weeks per quarter, I have had to change things somewhat from my initial goal. My project is based on my research interests, which are high-performance simulations of magnetohydrodynamics (MHD). My initial proposal was to build a "simple" (notice the foreshadowing use of quotations) finite-element simulation code that solved the ideal MHD equations, and use it to model some relatively well-known physical problem like the interaction between the Earth's magnetic field and the solar wind. Of course, this being the first time in my life attempting (and taking coursework in) high performance computing, numerical methods for solving differential equations, and magnetohydrodynamics, such a project was decidedly *not* simple. I spent quite a bit of time in the first half of the quarter reading up on the numerical methods commonly used to solve equations like the MHD equations, settled on a scheme that seemed powerful yet relatively straightforward (the Kurganov-Tadmor high-resolution finite-volume scheme), and then got quite overwhelmed trying to implement the three-dimesnional extension of it with such a complicated set of equations as the MHD equations. I actually got a sequential version of it (somewhat) working, but could not get the magnetic field portion to behave properly with the energy density in my simulation tests.

With deadlines looming on the horizon, I decided to simplify things significantly on the physical and numerical side of things by eliminating the magnetic field portion of the equations and using a more straightforward numerical scheme. This allowed me to focus more on the actual point of the project, which was to implement the HPC techniques we have learned throughout the quarter. So, the content of this report will describe the results of this simplified version – which still took quite a bit of effort and taught me quite a lot along the way! The application I have built is a parallel solver for the compressible, inviscid, force-free Navier-Stokes equations in conservative form, written in Fortran 90 with MPI for parallelization and Parallel netCDF for I/O. The simulation domain is a three-dimensional box with periodic boundary conditions, and the equations are integrated forward in time using the Rusanov scheme. In recognition of the great pains I went through to develop, test, and debug this code, as well as the fact that it is a parallel solver for the inviscid Navier-Stokes equations, I have named the application `PaINS`. In this report, Section 2 describes the design overview of `PaINS` and explains the strategy behind the design choices I have made, and Section 3 presents the physical results and application performance of various test runs.

2 PaINS design overview

2.1 Physical model

The compressible and inviscid form of the Navier-Stokes equation, along with equations for conservation of mass and energy, are of the form

$$\frac{D\rho}{Dt} = -\rho(\nabla \cdot \boldsymbol{u}) \tag{1}$$

$$\frac{D\boldsymbol{u}}{Dt} = -\frac{\nabla p}{\rho} + \boldsymbol{F} \tag{2}$$

$$\frac{DE}{Dt} = -\frac{p}{\rho}(\nabla \cdot \boldsymbol{u}), \tag{3}$$

where D/Dt represents the material (advective) derivative, ρ is the fluid density, $\boldsymbol{u}$ is the fluid velocity, E is the energy density, $\boldsymbol{F}$ is the net body force acting on the fluid (e.g. gravity), and p is the fluid pressure given by

$$p = (\gamma - 1)\Big(E - \frac{1}{2}\rho u^2\Big), \tag{4}$$

with γ being the ratio of specific heats of the fluid and u^2 being the square of the velocity magnitude. In finite-element numerical schemes, it is convenient to express these equations in a conservative form, which entails expressing the time rate of change of a quantity being equal to a spatial flux of that quantity at every point in space. This way, the fluxes into and out of each finite element cell can be tracked and used to update the values within the cell. Thus, the conservative form of (1)-(3), setting $\boldsymbol{F} = 0$, is

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{5}$$

$$\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u} + p\boldsymbol{I}) = 0 \tag{6}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot \Big[\boldsymbol{u}(E + p)\Big] = 0. \tag{7}$$

In equation (6), the quantity $\boldsymbol{u}\boldsymbol{u}$ is the tensor product of the velocity vector, and the quantity $\boldsymbol{I}$ is the second-rank dyadic tensor corresponding to the identity matrix. Taking the divergence of these second-rank tensors reduces their dimensionality by one, which completes the vector equation. Equations (5)-(7), along with the expression for p in (4), are the equations that are modeled by `PaINS`.

2.2 Numerical scheme

The numerical scheme in `PaINS` is the Rusanov scheme, which is a finite-element method designed for conservative equations. Spectral schemes are much more accurate than finite

element schemes and are thus pretty much the status quo in many HPC fluids fields, but they are generally much more complicated to implement, so I decided to stick with a finite element scheme. However, another interesting comparison between the two is between their scale-up: despite the higher accuracy of spectral schemes, they incur a larger communication cost because they require Fourier transforms and thus all-to-all communication at each timestep, while finite-element equations, despite being less accurate, scale better in massively parallel settings. As the HPC field moves towards the exascale generation, some people think that in some applications it may actually be advantageous to revert to finite-element schemes. Of course, I don't expect `PaINS` to be run at such huge scales (I just chose a finite-element scheme for its simplicity), but I was interested by the idea!

Equations of the form

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0 \tag{8}$$

are called scalar hyberbolic conservation laws. As mentioned earlier, these equations interface nicely with finite-element codes since they can explicitly enforce conservation of quantities (up to machine accuracy) by updating the values in each element according to the spatial flux into and out of the cell. `PaINS` is a three-dimensional code, so the relevant generalization of the conservation law (8) is

$$\frac{\partial u}{\partial t} + \nabla \cdot (\mathbf{\Phi}(u)) = \frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} + \frac{\partial g(u)}{\partial y} + \frac{\partial h(u)}{\partial z} = 0. \tag{9}$$

Equations (8) and (9) each define Riemann problems at each cell, which can be solved exactly at a large computational cost (this is the Godunov method). A much more efficient method is to approximate the solution to the Riemann problem in some convenient way, and there are many established schemes that do so. The Rusanov scheme approximates the Riemann solution by defining a numerical flux based on the local propagation speed of waves. The three-dimensional form of the Rusanov scheme, for a conserved scalar quantity u with a conservation law of the form (9), is

$$u^{n+1}_{j,k,l} = u^n_{j,k,l} - \Delta t \left(\frac{F^n_{j+\frac{1}{2},k,l} - F^n_{j-\frac{1}{2},k,l}}{\Delta x} + \frac{G^n_{j,k+\frac{1}{2},l} - G^n_{j,k-\frac{1}{2},l}}{\Delta y} + \frac{H^n_{j,k,l+\frac{1}{2}} - H^n_{j,k,l-\frac{1}{2}}}{\Delta z} \right), \tag{10}$$

where $u^n_{j,k,l}$ is the value of u at the nth time step at cell j,k,l, and F,G,H represent the numerical fluxes into and out of the cell in each direction. These fluxes are defined by

$$F^n_{j+\frac{1}{2},k,l} = \frac{f(u^n_{j,k,l}) + f(u^n_{j+1,k,l})}{2} - \frac{\max(|f'(u^n_{j+1,k,l})|, |f'(u^n_{j,k,l})|)}{2}(u^n_{j+1,k,l} - u^n_{j,k,l}) \tag{11}$$

$$G^n_{j,k+\frac{1}{2},l} = \frac{g(u^n_{j,k,l}) + g(u^n_{j,k+1,l})}{2} - \frac{\max(|g'(u^n_{j,k+1,l})|, |g'(u^n_{j,k,l})|)}{2}(u^n_{j,k+1,l} - u^n_{j,k,l}) \tag{12}$$

$$H^n_{j,k,l+\frac{1}{2}} = \frac{h(u^n_{j,k,l}) + h(u^n_{j,k,l+1})}{2} - \frac{\max(|h'(u^n_{j,k,l+1})|, |h'(u^n_{j,k,l})|)}{2}(u^n_{j,k,l+1} - u^n_{j,k,l}) \tag{13}$$

Equations (10)-(13) form the Rusanov scheme, and are implemented in `PaINS` to advance forward in time the values of the 5 scalar quantities in (5)-(7), namely the density ρ, energy density E, and x, y, z components of the velocity $\boldsymbol{u}$. As with all fluid codes, the velocity of the fluid must be tracked carefully so that the time step size is chosen such that the CFL condition is satisfied. The CFL condition for this scheme is

$$\max\left(\frac{\max_j |f'(u^n_{j,k,l})|}{\Delta x}, \frac{\max_k |g'(u^n_{j,k,l})|}{\Delta y}, \frac{\max_l |h'(u^n_{j,k,l})|}{\Delta z}\right)\Delta t \leq \frac{1}{2}. \tag{14}$$

2.3 Domain decomposition

The simulation domain is decomposed across the compute nodes one-dimensionally in the z-direction. This splits the original domain, which is box-shaped, into a number of "slabs" stacked on top of each other, with one process per slab (note the slabs are not two-dimensional, they are still three-dimensional boxes but with the z-axis having a length much shorter than the original). The z-direction was chosen because Fortran is a column-major language, so slices in the z-direction of a three-dimensional array are contiguous in memory and we can exploit data locality more. This data locality helps cache performance especially when sending information in between ghost cells and writing results out to disk. Since the domain is split along the z-direction, `PaINS` performs best when the longest dimension of the original simulation box is in the z-direction.

The Rusanov scheme (10)-(13) requires that each individual volume element within the array be able to access its two nearest neighbors in each direction in order to update correctly at each time step. This is taken care of in the x and y directions automatically (even at the edges of the box via periodic boundary conditions), but since each MPI process has its own address-space, the data for ghost cells at the z-boundaries for each slab must be sent/received to/from the appropriate processes. The advantage of the one-dimensional parallelization is that it reduces the number of messages necessary per cell since only the top and bottom of the slab need to be accounted for. Since the boundary conditions are periodic, this forms a ring topology among the processes. A diagram depicting the ghost cell exchange is given in Figure 1. The boundary "cells" are actually two-dimensional slabs of cells in the x and y directions, and the ghost cell data must be sent for all five scalar physical quantities of the simulation. To reduce the number of messages, the data for all five quantities is packed into one array for the previous neighboring process and one array for the next neighboring process, then unpacked and stored properly after being received. Since there are two layers of boundary cells for each rank, the minimum thickness of each slab in the z-direction must be greater than or equal to 3, meaning the maximum number of processes for a given run should be no more than the number of elements in the z-direction divided by 3. Thus, while simple and easy to implement, this one-dimensional parallelization structure somewhat limits the scalability for a fixed resolution size.

The CFL condition (14) is a global condition, so if we are to ensure temporally coherent output from the parallelized code we must call `MPI_reduceAll()` at the start of each time step to compute the maximum across the entire domain. From that maximum, the size of

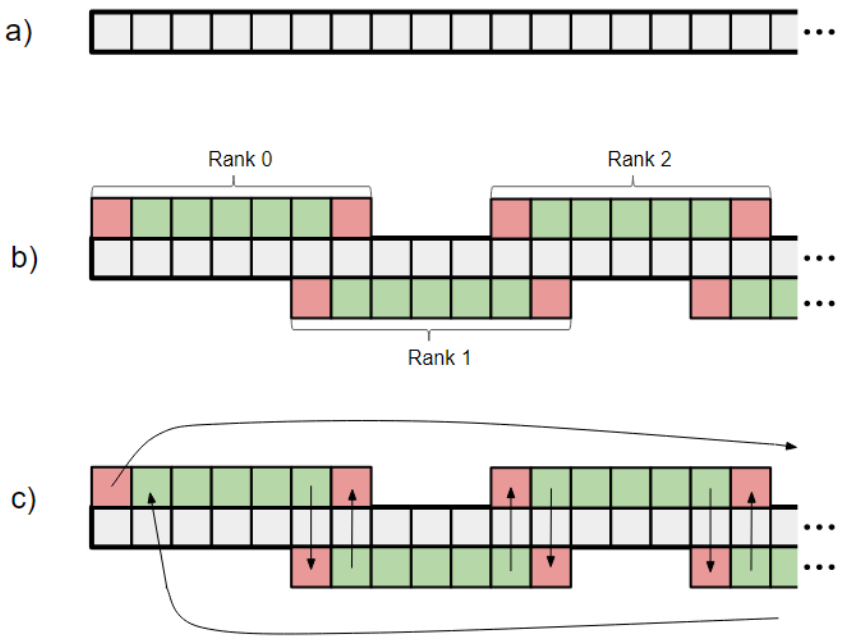

Figure 1: Schematic of the one-dimensional parallelization and ghost cell exchange in `PaINS`. The original, full array (a) is decomposed into equal-sized chunks (b) according to the number of elements along the z-direction and the total number of processes. Updates are calculated according to the Rusanov scheme (10)-(13) for the interior cells (green), and not for the boundary cells (red). Then, the updates for each boundary cell are received from the upper and lower neighbors of each rank, while the computed updates for the neighbors' boundary cells are sent to the appropriate neighbor (c).

the next time step can be computed according to (14). In PaINS, as is common in some fluid codes, I chose to also multiply the CFL-computed time step by a "'safety-factor" of 0.9 just to ensure it never pushes the limit too much. Also, since we can increase the time step size in regions with low flow velocities, the next time step is always chosen to be the minimum between the CFL-computed value (which may be huge if the velocities are slow or zero), the previous time step increased by 5%, and a user-defined maximum time-step. This setup provides a balance between safety (the CFL condition is always satisfied, and we can be sure we aren't "jumping" too far over discontinuities) and performance (we can "speed up" the simulation whenever possible to do so).

2.4 I/O & visualization tools

I have written PaINS to support parallel I/O, since the data output size for three-dimensional simulations grows large very quickly when increasing the resolution (doubling resolution in each direction multiplies the necessary storage space by 8). This was done via the Parallel netCDF library, which handles all of the parallel I/O details under the hood and is well-documented online. With this library, the user can also pre-compute the sizes of the data writes each process will make and adjust the file-system stripe settings to optimize the I/O portion. The output file contains the coordinate axes data, user-specified units, and the simulation output data for each of the five quantities at each time step that was saved. The user sets the parameter `save_n_steps` in the `model_params.f90` module to choose how much the simulation saves the data. At each time step that writes data, all the processes write their portions of the full array (truncating the boundary cells in the z-direction to avoid redundancy, except for the first and last rank) in collective mode to a single shared netCDF file.

A nice benefit of saving the output arrays into a netCDF file is that the `.cdf` format is one that is recognized by the powerful analysis and visualization software tool `VisIt`. This tool allows the user to run a parallel compute engine (either locally or on a host machine) that generates plots, animations, and other analytical information about the data set. Doing the same thing in Python would be a lot more involved, and completely out of the question with Fortran, but as long as the user has access to an installation of `VisIt` (installed on Cori as well as Hyades) then this is a convenient "side-effect" capability of PaINS.

3 Test Results

There is a wide variety of test problems commonly used to evaluate the functionality and benchmark the performance of fluids codes. Since the Rusanov scheme is supposed to be able to handle shocks better than similar alternatives like the Lax-Friedrichs scheme, I chose to test PaINS with a simple blast wave. The parameters for this setup are as follows (note that although I included capability in PaINS to handle physical units, I ran these tests with arbitrary units). For the simulation box, x and y both run between -0.5 and 0.5, while z runs from -0.75 to 0.75. The initial density is constant everywhere initially, with $\rho = 1.0$, and the

velocity is zero everywhere initially. The pressure is set to 0.1 everywhere initially, except for a spherical region within $r \leq 1.0$ near the origin where $p = 10$ (100 times the ambient pressure). This creates a blast wave that propagates outwards from the center and "reflects" off the edges of the simulation box via the periodic boundary conditions.

All testing was done on Cori, although it should work fine on Hyades by modifying the appropriate commands in the `Makefile` and making sure the paths to the Parallel netCDF libraries are visible to the compiler driver. On Cori, compiling is straightforward – just do `module load cray-parallel-netcdf` and then `make` from the directory with all the source code. This creates the executable `PaINS_executable` which can then be submitted in a batch script or run from an interactive job.

3.1 Model output

A "production" test was done with a resolution of 200 elements each in the x- and y-directions and 300 elements for the z-direction. The simulation was run for a total of 1600 time steps to capture the dynamics of shockwaves interacting with themselves via reflection from the periodic boundary conditions. Output was saved every 40 steps, which led to a total output size of 19.2 GB (all quantities are stored in double precision, which probably isn't necessary). Using `VisIt`, this output was visualized in several plots shown here. Figure 2 shows a "three-slice" of the density field (at $x = 0$, $y = 0$, and $z = 0$) in the simulation at various time steps, and Figure 3 shows a density slice in the x-z plane at various time steps for greater detail. Because of the symmetry of the initial conditions and the periodic boundary conditions, we should expect reflectional symmetry in the patterns about $x = 0$, $y = 0$, $z = 0$, which is indeed the case. The results make physical sense, as they clearly show the blast wave propagating outwards, reflecting off of the boundaries and forming complex patterns as the reflected density waves self-interact.

3.2 Performance & scaling characteristics

The strong scaling was tested by running the simulation for a box of size $100 \times 100 \times 150$ for a total of 800 time steps, writing a total of 10 time steps each run to take into account the parallel I/O performance in the scaling results. This was done in the sequential version, which took around 20 minutes to complete, and then for the parallel version with 2, 4, 8, 16, 24, and 48 processes. Figure 4 shows the strong scaling characteristics of `PaINS` with the aforementioned parameters. Clearly, the performance increases as the number of processes increases, but this is by no means an "ideal speedup" (these results are about halfway between "not good" and "so-so" in the sample scaling plot given in the lecture notes for performance theory).

I think that the most obvious reason for this is that the one-dimensional parallelization fundamentally limits how many processes can be used for a fixed problem size (see section 2.3), so the scale-up is close to linear only briefly before the performance ceiling becomes significant. When the number of processes gets large, each process is doing computations

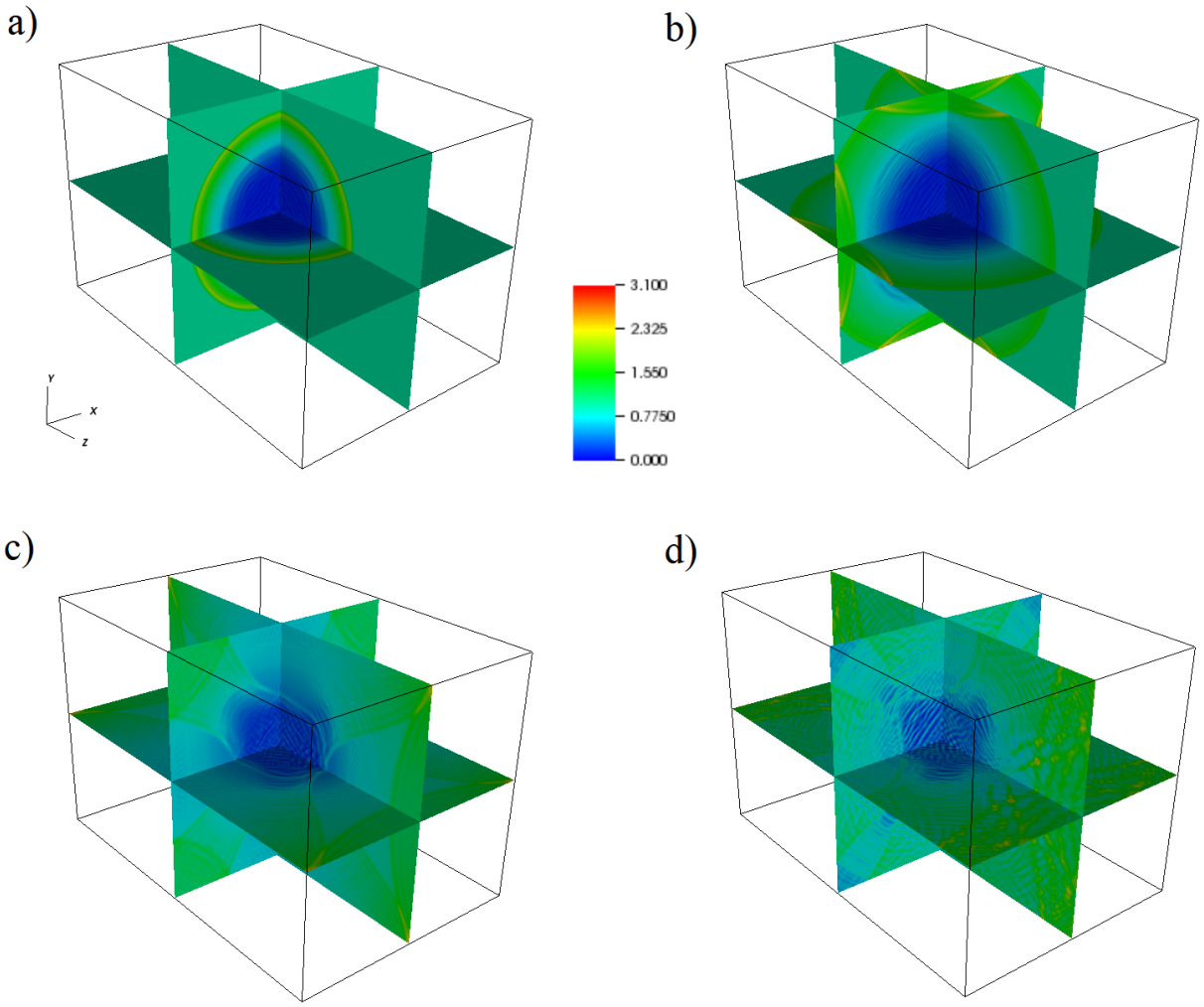

Figure 2: "Three-slice" of the density field in the simulation for the blast wave test run. The blast wave spreads out from the center and nears the edge of the box in (a), then reflects first off of the x and y boundaries (b) and then the z boundary (c), forming complex shockwave self-interaction patterns (d) as the reflected density waves propagate through the box.

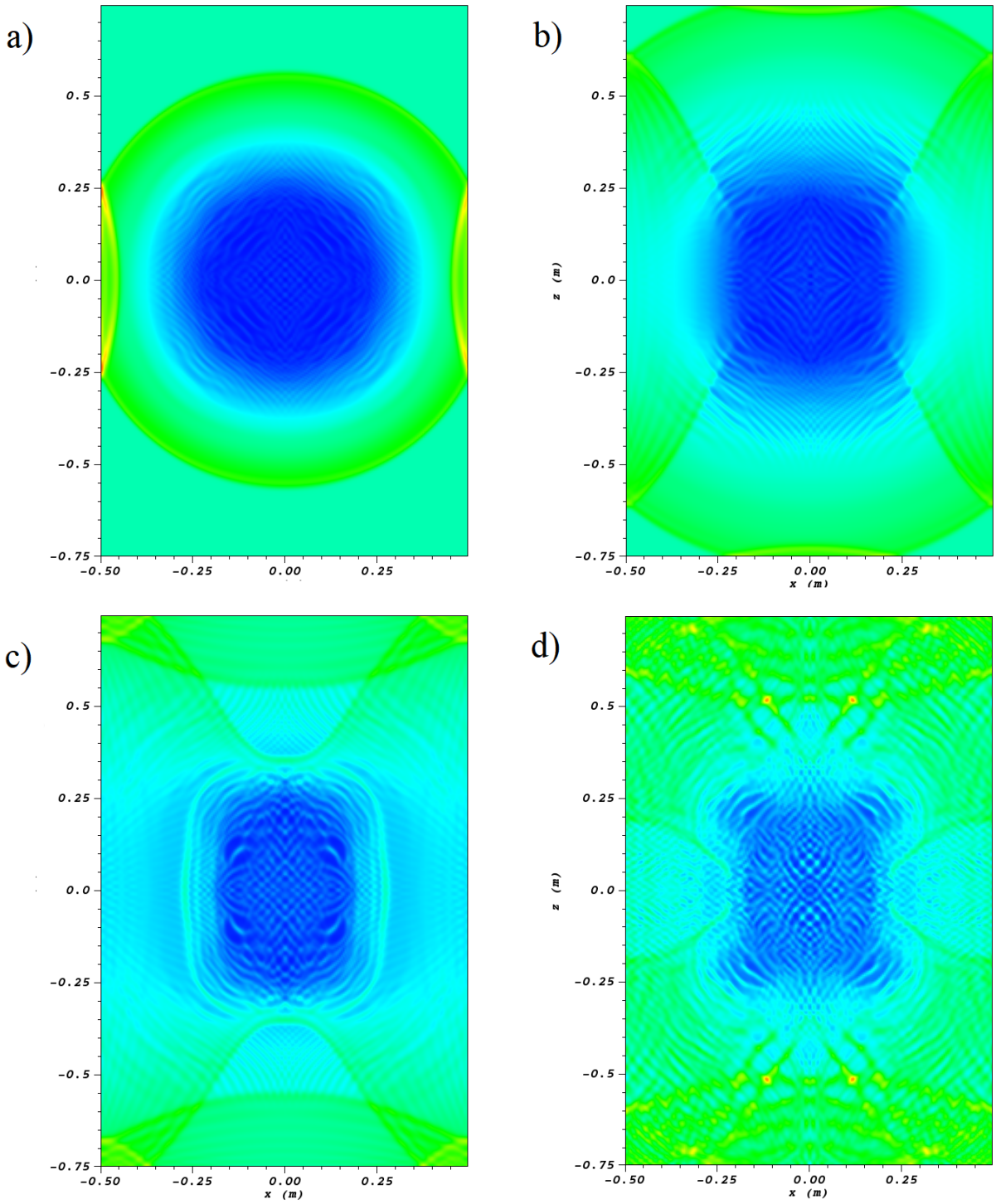

Figure 3: Slices of the density field in the x-z plane for the blast wave test run. The blast wave spreads out from the center and reflects first off of the x and y boundaries (a) and then the z boundary (b), forming complex shockwave self-interaction patterns (c), (d) as the reflected density waves propagate through the box. Same color scale as Figure 2.

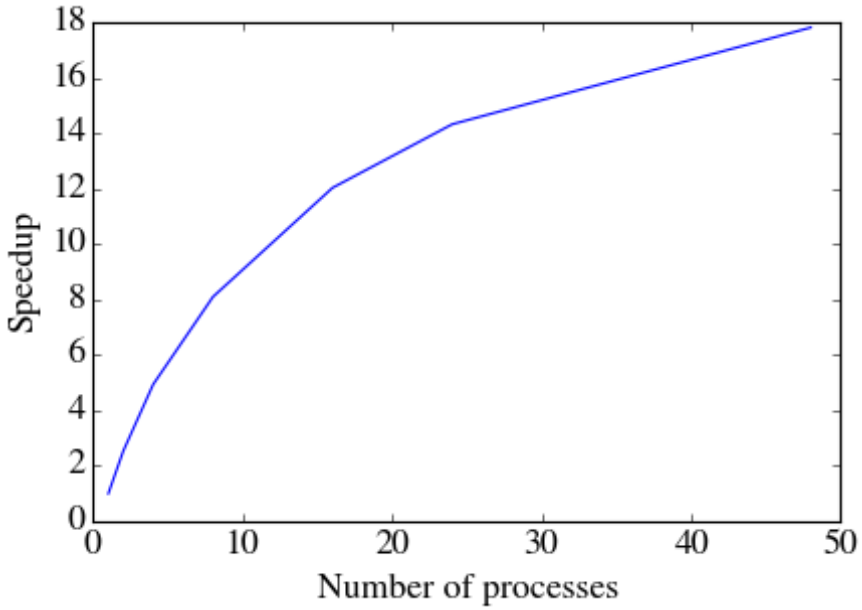

Figure 4: Strong scaling results for the parallelized `PaINS` with respect to the sequential version. These were done for a box of size $100 \times 100 \times 150$ for a total of 800 time steps, writing a total of 10 time steps, with 2, 4, 8, 16, 24, and 48 processes.

for a very thin slab of the simulation domain, but the dimensions of the x and y directions are the same per process with respect to the sequential problem. Thus, the size of the ghost cell layers being sent between neighboring processes are of the same order as that of the subdomain being computed by each process, which is not efficient. A more detailed look into the timing reveals this to be the case – Figure 5 shows the percentage of runtime taken up by communication and by the actual numerical calculations, and clearly, the communication time starts to dominate as we near the maximum number of processes.

Parallelizing along the y direction as well would definitely help with this issue, as it would further partition the domain (this would be the "pencil" decomposition) and allow for more total processes to be used on a fixed problem size. This would come at the cost of increased communication requirements, since ghost cell data would need to be sent between neighboring processes for both interfaces in the z and y directions (a total of four two-dimensional arrays of numbers for all five physical quantities in the simulation). Another option which would enhance performance would be to rebuild `PaINS` as a hybrid MPI/OpenMP application, so that the domain could be partitioned as it is here but each process would be able to compute the updates for its own subdomain in parallel. This hybrid version would still be limited by the same issue of the problem size bounding the number of processes as is seen with the pure MPI version.

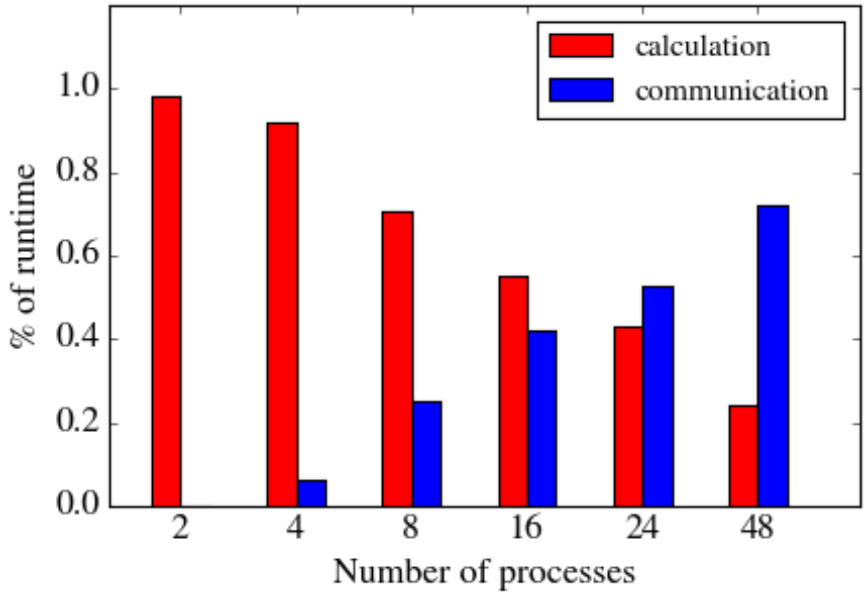

Figure 5: The percentage of runtime in `PaINS` taken by communication and calculation for the runs with 2, 4, 8, 16, 24, and 48 processes. Clearly, communication costs start to dominate as we approach the performance ceiling for the application.