CMSE 822 Final Project

Carolyn Wendeln & Gabe Appleton

December 13, 2021

1 Introduction

Modern astrophysical research has become increasingly dominated by large scale, multi-dimensional simulations. Phenomena such as supernovae core collapse, coronal mass ejections, or neutron star mergers all rely on multi-scale simulations. These simulations can span orders of magnitude in scale separation, yet these vast scales are required to resolve the key underlying physics involved. Although these large scale simulations have become a necessity, they often require an ever-increasing amount of computational resources.

One key feature that most of these astrophysical phenomena have in common is that they are often coupled to hyperbolic partial differential equations (PDEs). Equations such as Euler, Cauchy, or Navier–Stokes are often used for simulating astrophysical plasmas. These simulations often call for techniques such as adaptive mesh refinement (AMR) and high order finite difference or finite volume methods.

For this research project, we will be investigating second and fourth order discretization of hyperbolic PDEs. In particular, we will look at the two dimensional form of the Advection Equation, Inviscid Burgers' Equation, and Viscous Burgers' Equation. We chose these equations because they are generally easier to solve than Euler, Cauchy, and Navier-Stokes. This work will focus on these equations over a periodic domain on a uniform mesh grid using finite difference methods.

The goal of this research project is to understand the underlying issues surrounding hyperbolic solvers for fluid equations. I will investigate various optimization techniques for a single node. My work will begin with parallelization using `OpenMP` to target CPUs. After parallelisation, we will employ other techniques such as cache blocking and loop scheduling to see the effects of strong and weak scaling.

2 Methodology

2.1 Finite Difference Methods for Hyperbolic PDEs

For this research project, we look to solve and parallelize three PDEs. The first is the linear advection equation, which can be described as

$$\frac{\partial a}{\partial t} + u\frac{\partial a}{\partial x} = 0 \tag{1}$$

where $a(x,t)$ is a scalar quantity and u is the velocity at which it is advected. For our research project, we chose to make $a(x,t)$ a Gaussian function of the form

$$a(x,t) = a_0(t)e^{-50(x-0.5)^2} \tag{2}$$

To solve this PDE, we use a finite-difference discretization in which the domain is divided into a sequence of points where we store the solution. We then solve this PDE numerically by discretizing the solution at these points.

Let the index i denote the point's location, and a_i denote the discrete value of $a(x)$ at point i. Figure 1 shows the grid. To discretize in time, we denote the time-level with a superscript n, such that $a_i^n = a(x_i, t_n)$. For a fixed Δt, a time level n corresponds to a time of $t = n\Delta t$. A simple first-order accurate centered-difference discretization is then

$$\frac{a_i^{n+1} - a_i^n}{\Delta t} + u\frac{a_{i+1}^n - a_{i-1}^n}{2\Delta x} = 0 \tag{3}$$

This is a forward-time, centered-space explicit method, since the new solution, a_i^{n+1}, depends only on information at the old time level, n. When implemented into `C++`, we are iterating through time steps n and displacement steps i to solve for a_i^{n+1}

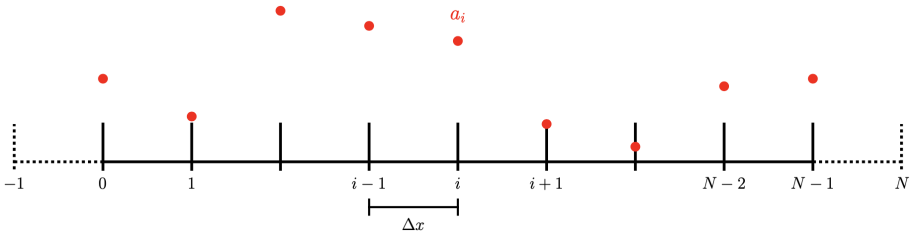
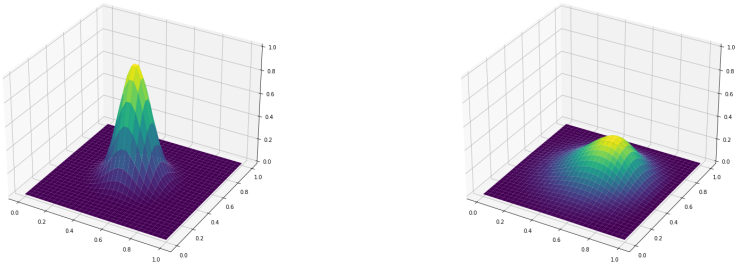


Figure 1: A simple finite-difference grid [Zingale, 2020]. The solution is stored at each of the labeled points. The dotted lines show the ghost points used to extend our grid past the physical boundaries to accommodate boundary conditions. In our instance, we assume a periodic domain, so the points 0 and $N-1$ are the same physical point in space.



(a) Time step 0 (b) Time step 1000

Figure 2: Visualization of 2D Viscous Burgers' Equation

The second equation we wish to investigate is Inviscid Burgers' equation. This equation is a nonlinear hyperbolic equation, and given as:

$$\frac{\partial a}{\partial t} + a\frac{\partial a}{\partial x} = 0 \tag{4}$$

Here $a(x,t)$ is both the quantity being advected and the speed at which it is moving. The centered-difference discretization for Inviscid Burgers' equation is then

$$\frac{a_i^{n+1} - a_i^n}{\Delta t} + a_i^n(\frac{a_{i+1}^n - a_{i-1}^n}{2\Delta x}) = 0 \tag{5}$$

The third and final equation we wish to investigate is Viscous Burgers' equation

$$\frac{\partial a}{\partial t} + a\frac{\partial a}{\partial x} = \nu\frac{\partial^2 a}{\partial x^2} \tag{6}$$

Here ν is the viscosity. The centered-difference discretization for Viscous Burgers' equation is then

$$\frac{a_i^{n+1} - a_i^n}{\Delta t} + a_i^n(\frac{a_{i+1}^n - a_{i-1}^n}{2\Delta x}) = \nu(\frac{a_{i+1}^n - 2a_i^n - a_{i-1}^n}{\Delta x^2}) \tag{7}$$

2.2 Parallization Techniques

For this research project we aim to parallize our code using `OpenMP`. Our main interest is speeding up the major for loop in our code which iterates through the bulk of our spatial domain. The boundaries of our domain are not included in this for loop, for they have specific boundary conditions that must be met

to maintain periodicity. For example, the major for loop of our 1D Advection code appear as follows:

```
1  for(int t=0; t<n; ++t)
2  {
3      double start = omp_get_wtime();
4
5      #pragma omp parallel for num_threads(thrd_cnt)
6
7      for(int i=1; i<(N-2); ++i)
8      {
9          a_write[i] = a_read[i] -
10         (((u * delta_t) / (2 * delta_x))*(a_read[i+1] - a_read[i-1]));
11     }
12
13     double end = omp_get_wtime();
14     double time_taken = end-start;
15
16     loop_time[t] = time_taken;
17 }
```

In addition to parallizing our code using `#pragma omp parallel`, we are interested in seeing the effect on computation time if we use a loop scheduler. Loop schedulers are a helpful way to assign loop iterations to the number of threads available. The three types of loop schedulers we are interested in are Static, Dynamic, and Guided.

Static scheduler assigns one consecutive block of iterations to each thread. With static, the compiler determines the assignment of loop iterations to the threads at compile time. Static allows for a user specified chunk size for the size of blocks. If all iterations take roughly the same amount of time, static is the most efficient use of a loop scheduler.

Dynamic scheduler assigns blocks of iterations in a task queue, where threads will take one of these tasks whenever it finishes with its previous task. Similar to static, dynamic allows for a specified chunk size. Dynamic is most helpful when the iterations take different amounts of time to execute.

Lastly, guided scheduler gradually decreases the chunk size. It does so under the assumption that large chunks carry the least overhead, but smaller chunks are better for load balancing. The various schedules are illustrated in Figure 3.

For this research project, we wish to look at parallization for 2D Advection, 2D Inviscid Burgers, and 2D Viscous Burgers. In particular, we wish to look at strong and weak scaling, while incorporating cache blocking by loop scheduling. A detailed summary can be found in Table 1

3 Results & Discussion

3.1 Cache Blocking

For the three loop schedulers, we looked at the effect of chunk size in relation to the time needed to run for N = 512, n = 10,000, threads = 8, and the chunk size varied between 1 through 100. The results are seen in Figure 4. Both Static

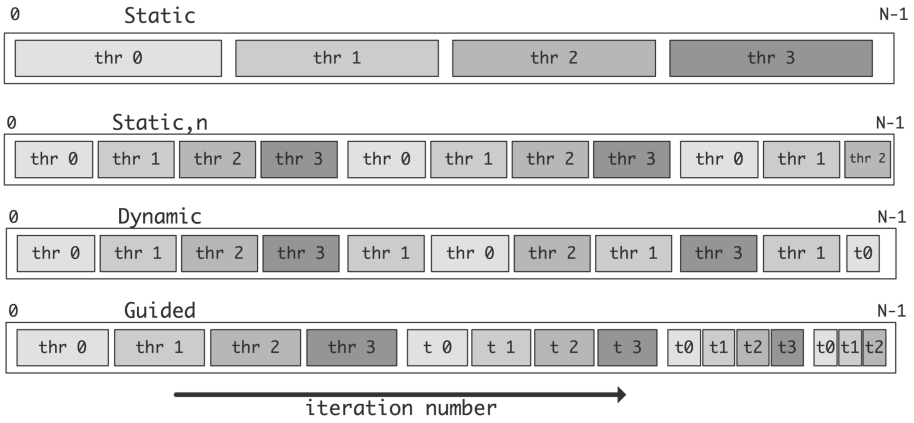


Figure 3: Illustration of the scheduling strategies of static, dynamic, and guided loop schedulers [Eijkhout, 2021].

and Dynamic scheduling have a minimum time around a chunk size of 32 and 64. This is likely due to HELP

3.2 Loop Schedules

4 Acknowledgments

C. W. would like to thank Dr. Andrew Christlieb for his invaluable knowledge on numerical solutions to PDEs.

References

[Eijkhout, 2021] Eijkhout, V. (2021). *Parallel Programming for Science and Engineering*.

[Zingale, 2020] Zingale, M. (2020). *Introduction to Computational Astrophysical Hydrodynamics*. The Open Astrophysics Bookshelf.

	Type of Scaling	Scheduler Type	Number of x,y Steps (N)	Number of Time Steps (n)	Number of Threads (t)
2D Advection	strong	none	32 - 2,048	10,000	1 - 8
		static	32 - 2,048	10,000	1 - 8
		dynamic	32 - 2,048	10,000	1 - 8
		guided	32 - 2,048	10,000	1 - 8
	weak	none	32 - 2,048	1000N ÷ 32	1 - 8
		static	32 - 2,048	1000N ÷ 32	1 - 8
		dynamic	32 - 2,048	1000N ÷ 32	1 - 8
		guided	32 - 2,048	1000N ÷ 32	1 - 8
2D Burgers	strong	none	32 - 2,048	10,000	1 - 8
		static	32 - 2,048	10,000	1 - 8
		dynamic	32 - 2,048	10,000	1 - 8
		guided	32 - 2,048	10,000	1 - 8
	weak	none	32 - 2,048	1000N ÷ 32	1 - 8
		static	32 - 2,048	1000N ÷ 32	1 - 8
		dynamic	32 - 2,048	1000N ÷ 32	1 - 8
		guided	32 - 2,048	1000N ÷ 32	1 - 8
2D Viscous Burgers	strong	none	32 - 2,048	10,000	1 - 8
		static	32 - 2,048	10,000	1 - 8
		dynamic	32 - 2,048	10,000	1 - 8
		guided	32 - 2,048	10,000	1 - 8
	weak	none	32 - 2,048	1000N ÷ 32	1 - 8
		static	32 - 2,048	1000N ÷ 32	1 - 8
		dynamic	32 - 2,048	1000N ÷ 32	1 - 8
		guided	32 - 2,048	1000N ÷ 32	1 - 8

Table 1: Summary of tasks to run. Note that N is only run for powers of 2.

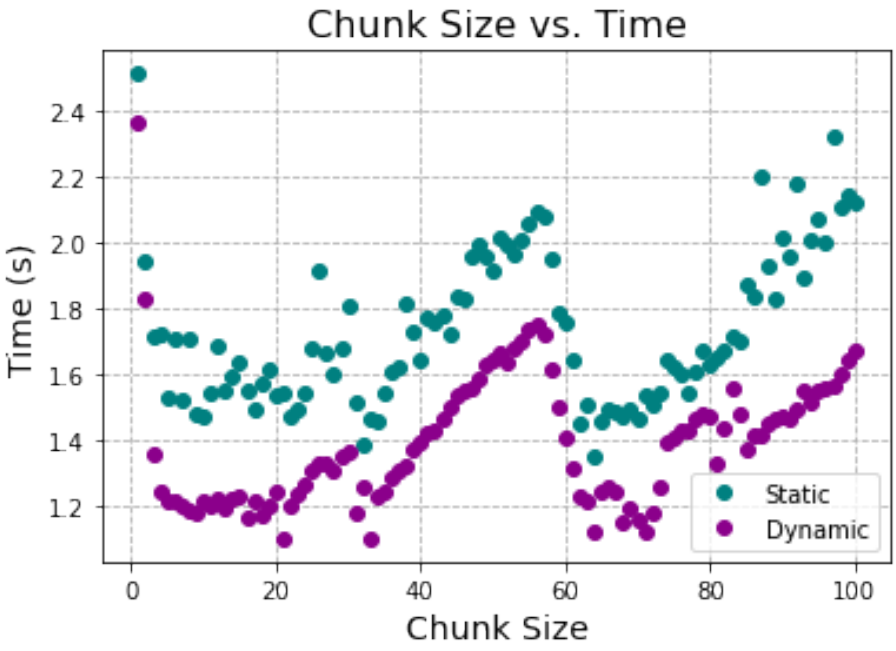
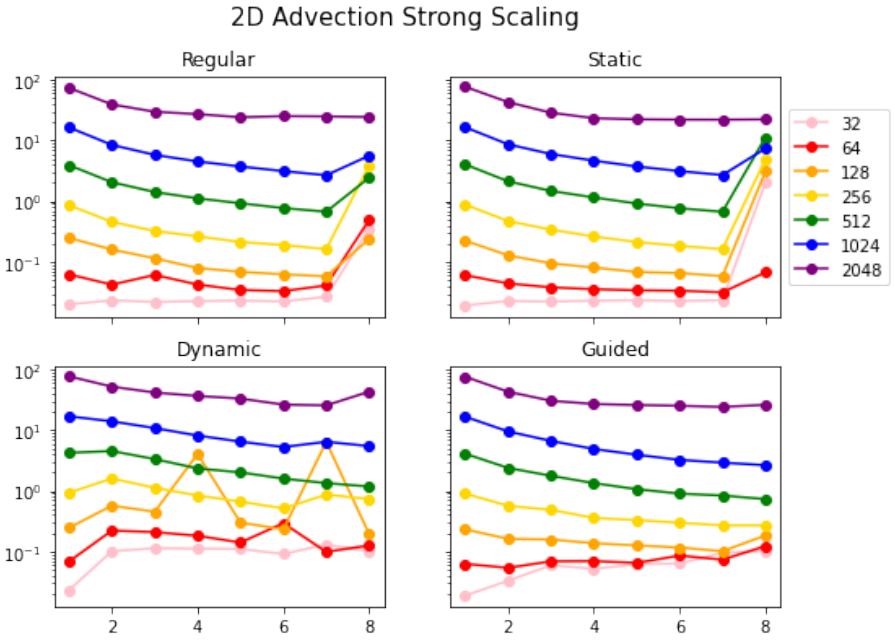
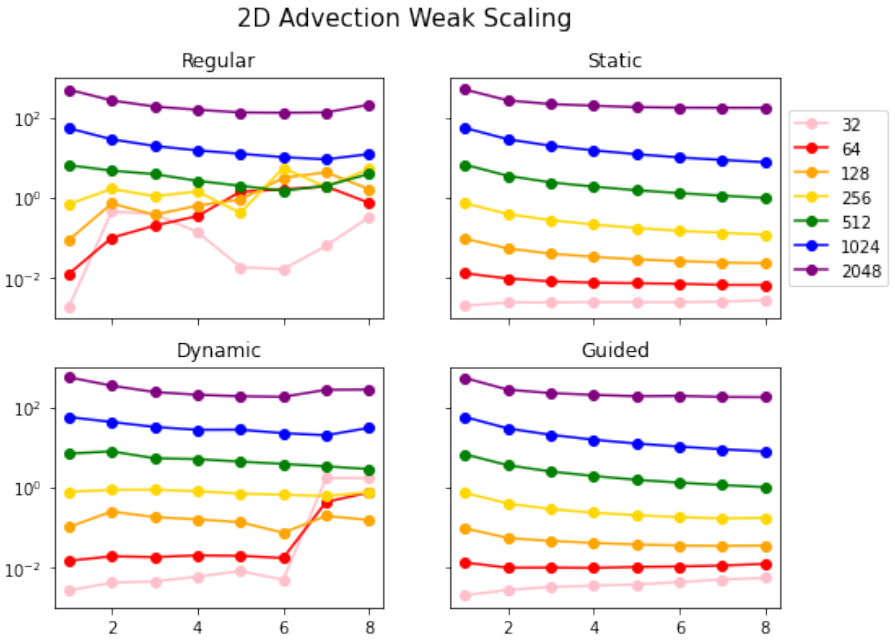


Figure 4: Chunk.

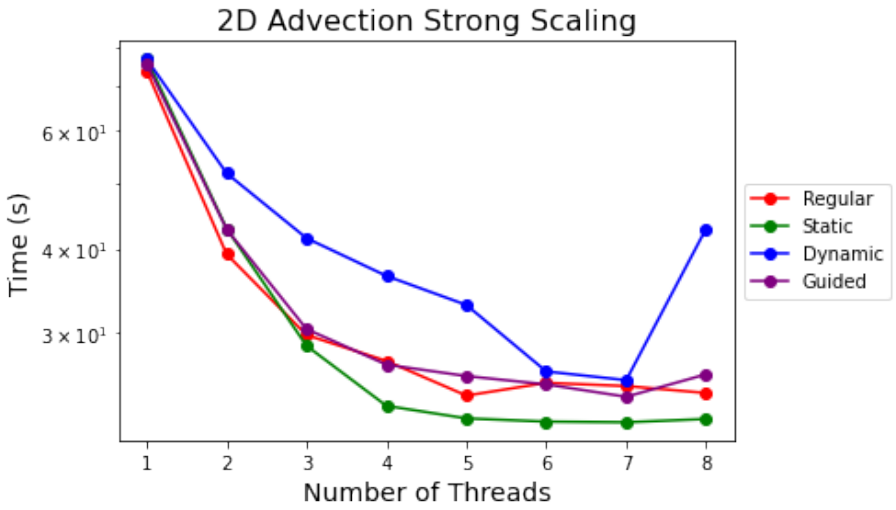


(a)

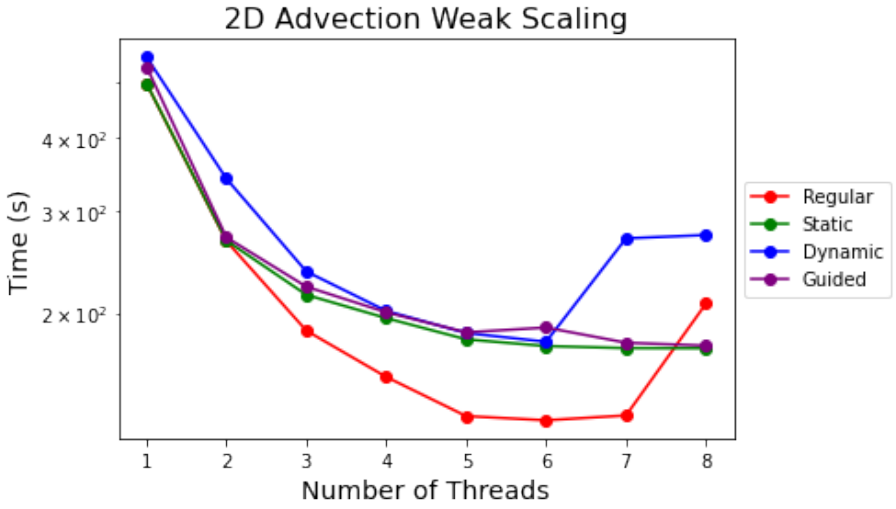


(b)

Figure 5: 2D Advection



(a)



(b)

Figure 6: 2D Advection