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 such multi-scale simulations. A common theme throughout the hardest of these simulations is the tight coupling of the different underlying physical phenomena involved over broad scales in both length and time. Although these multi-scale simulations have become a necessity to increasing our knowledge, 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 nonlinear hyperbolic partial differential equations (PDEs). Equations such as Euler, Cauchy, or Navier–Stokes are often used for simulating astrophysical plasmas. The Simulation of these equations often calls for techniques such as adaptive mesh refinement (AMR), high order finite difference, or high order 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 compared to Euler, Cauchy, and Navier-Stokes. This work will focus on these three 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. We will investigate various optimization techniques for a single node. Our 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 commonly used for simulating fluid dynamics. The first is the linear Advection Equation. This equation can be described as follows:

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

where $a(x,t)$ is a scalar quantity for a fluid 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. For this method, the spatial domain is divided into a sequence of points where the solution of $a(x,t)$ is stored. The PDE is then solved numerically by discretizing the solution at these points. Figure 1 shows a representation of the discretized grid. Let the index i denote the point's location, and a_i denote the discrete value of $a(x)$ at point i. 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 first-order accurate discretization centered around the point i is then given as:

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

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 imple-

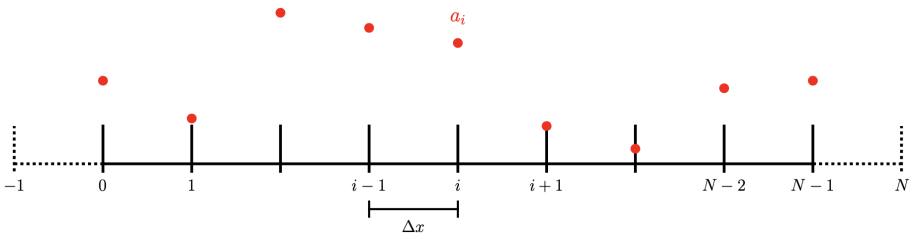


Figure 1: A simple finite-difference grid [Zingale, 2020]. The solution is stored at each of the labeled points. The dotted lines show the ghost points used to extend our grid past the physical boundaries to accommodate boundary conditions. In this instance, we assume a periodic domain, so the points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

mented into C++, we are iterating through time steps n and displacement steps i (i.e., a_i^n) to solve for a_i^{n+1}.

The second equation we wish to investigate is the Inviscid Burgers' Equation. This equation is a nonlinear hyperbolic equation, and is 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 the Inviscid Burgers' Equation is then:

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

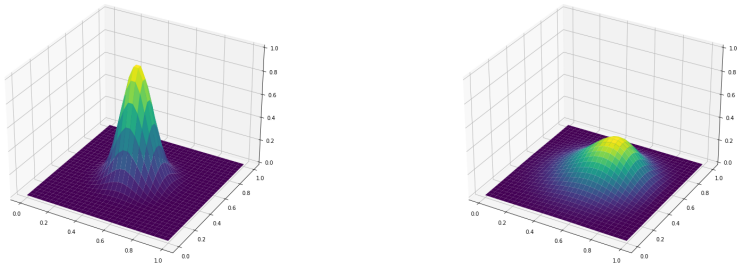
The third and final equation we wish to investigate is the Viscous Burgers' Equation. This equation is written as such:

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

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

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

A visual representation of the Viscous Burgers Equation can be seen in Figure 2. For this visualization, our x and y domain is between the values of 0 and 1 on a 512×512 grid. The centered-difference discretization for the Viscous Burgers' Equation was used to solve up to $t = 0.3$ seconds for $n = 1,000$ time steps. A time of $t = 0.3$ was chosen to prevent the formation of a shock.



(a) Time step $n = 0$ (b) Time step $n = 1000$

Figure 2: Visualization of the 2D Viscous Burgers' Equation

2.2 Parallization Techniques

For this research project we aim to parallize our code using `OpenMP`. Our main interest is speeding up the major loop which iterates through the bulk of our spatial domain. The boundaries of our domain are not included in this loop, as they have specific boundary conditions that must be met to maintain periodicity. As an example, the major loop of our 1D Advection code is as follows:

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

In addition to parallelizing 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 for this study are static, dynamic, and guided.

The static scheduler assigns a consecutive block of iterations to each thread. If a run has t threads, each thread will be assigned $\frac{1}{t}$ of the iterations. Static allows for a user specified chunk size for the size of blocks. If all iterations take roughly the same amount of time, static may be the most efficient loop scheduler.

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

Lastly, the 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 parallelization of the 2D Advection Equation, 2D Inviscid Burgers' Equation, and 2D Viscous Burgers' Equation. In particular, we wish to look at strong and weak scaling, while incorporating cache blocking and loop scheduling. A detailed summary of our work can be found in Table 1.

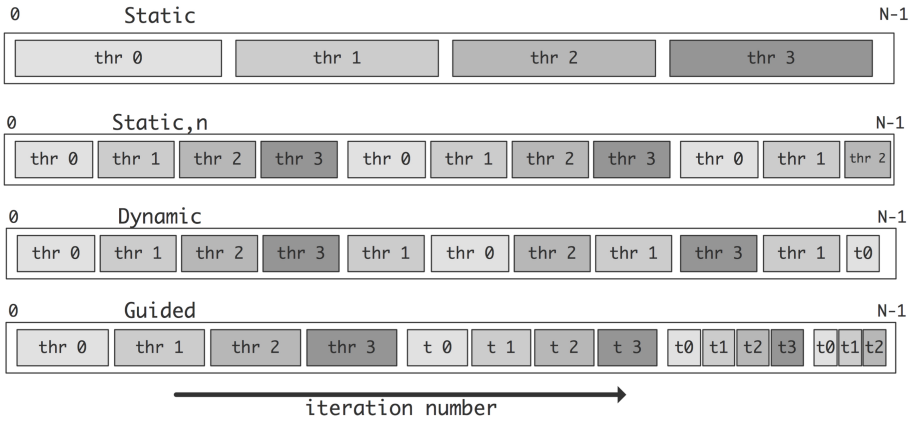


Figure 3: Illustration of the scheduling strategies of static, dynamic, and guided loop schedulers [Eijkhout, 2021].

3 Results & Discussion

3.1 Cache Blocking

For the dynamic and static loop schedulers, we looked at the effect of chunk size on the time needed to run for the following setup: $N = 512$, $n = 10,000$, $thrd\_cnt = 8$, and the chunk size varied from a value of 1 up to a value of 100. The results are seen in Figure 4. Both static and dynamic scheduling have minimum times around a chunk size of 32 and 64. We believe the pattern displayed follows the formula

$$\left\lceil \frac{N = 512}{chunk\ size} \right\rceil \div thrd\_cnt \tag{8}$$

With a thread count of 8, this formula gives us integer values at local minima of 22, 32, and 64.This would follow a pattern such that at these minima every thread gets the same number of chunks, therefore all threads run at the same wall clock time. For other chunk sizes, some threads will carry more workload, some will carry less, thus producing uneven wall clock times and increasing overall computation time.

3.2 Loop Scheduling

To investigate the effects that loop schedulers have on computation time, we looked at the effects of strong and weak scaling for a varying number of displacement steps, time steps, and thread count. For strong scaling, the time step was held constant at $n = 10,000$, while the thread count increased from $thrd\_cnt = 1$ through $thrd\_cnt = 8$, and the displacement steps increased by powers of 2 from $N = 32$ through $N = 2048$.

	Type of Scaling	Scheduler Type	Number of x,y Steps (N)	Number of Time Steps (n)	Number of Threads ($thrd\_cnt$)
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	$1000N \div 32$	1 - 8
		static	32 - 2,048	$1000N \div 32$	1 - 8
		dynamic	32 - 2,048	$1000N \div 32$	1 - 8
		guided	32 - 2,048	$1000N \div 32$	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	$1000N \div 32$	1 - 8
		static	32 - 2,048	$1000N \div 32$	1 - 8
		dynamic	32 - 2,048	$1000N \div 32$	1 - 8
		guided	32 - 2,048	$1000N \div 32$	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	$1000N \div 32$	1 - 8
		static	32 - 2,048	$1000N \div 32$	1 - 8
		dynamic	32 - 2,048	$1000N \div 32$	1 - 8
		guided	32 - 2,048	$1000N \div 32$	1 - 8

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

For weak scaling, the thread count increased from $thrd\_cnt = 1$ through $thrd\_cnt = 8$, the displacement steps increased by powers of 2 from $N = 32$ through $N = 2048$, and the time step increased proportionally to N by a factor of 1000/32 for values of $n = 1000$ up to a value of $n = 64,000$. The results for all of these runs for 2D Advection, Inviscid Burgers, and Viscous Burgers can be found in Figures 5, 6, and 7.

Generally there is a trend that when you increase the number of threads, computation time decreases. Likewise, as the displacement mesh increases, the computation time increases. This trend is seen for both strong and weak scaling. However there are a few notable exceptions that we will outline below.

The first of these is that when the workload distributed to each thread is relatively small, the benefit of increasing thread count is overwhelmed by overhead. You can see this clearly in Figure 5, especially in the runs without a scheduler and in the case of strong scaling, static advection. It is also fairly clear that this effect diminishes as the grid size increases, to the point where it

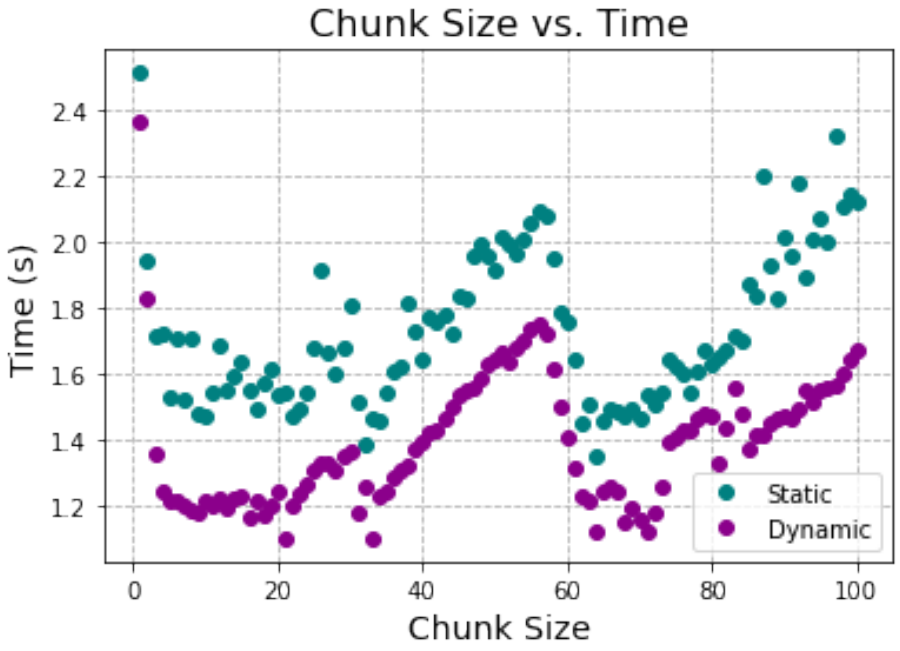


Figure 4: The effect of chunk size on computation time for schedule static and schedule dynamic.

is negligible in many runs, such as the guided runs.

The second major exception is found in the 2D weak scaling advection run in Figure 5, specifically the one with no scheduler. The case of grid size $N = 32$ appears to make sense to us. A major spike occurs when making the first thread, likely due to overhead in thread creation and memory management. This then decreases as you add threads up until 6, where it starts to reverse as threads are unable to find enough work to justify their creation. This, however, does not explain $N \in \{64, 128\}$ very well. Part of this is likely due to an interfering factor, as each doubling of N will cause an eight-fold increase in workload (double the width, double the length, double the time steps). To some extent this is an open question. A similar trend can be seen in the Inviscid Burgers weak scaling run with a static scheduler (Figure 6).

Figures 8, 9, and 10 show a detailed comparison of strong and weak scaling for various loop schedulers. These comparisons correspond to the $N = 2,048$ cases shown previously for the 2D Advection, Inviscid Burgers', and Viscous Burgers' Equations.

For the 2D Advection Equation shown in Figure 8, run time is minimized with no scheduler with ∼6-7 threads. The static scheduler works better for strong scaling, but only slightly, whereas no scheduler performs faster for weak scaling.

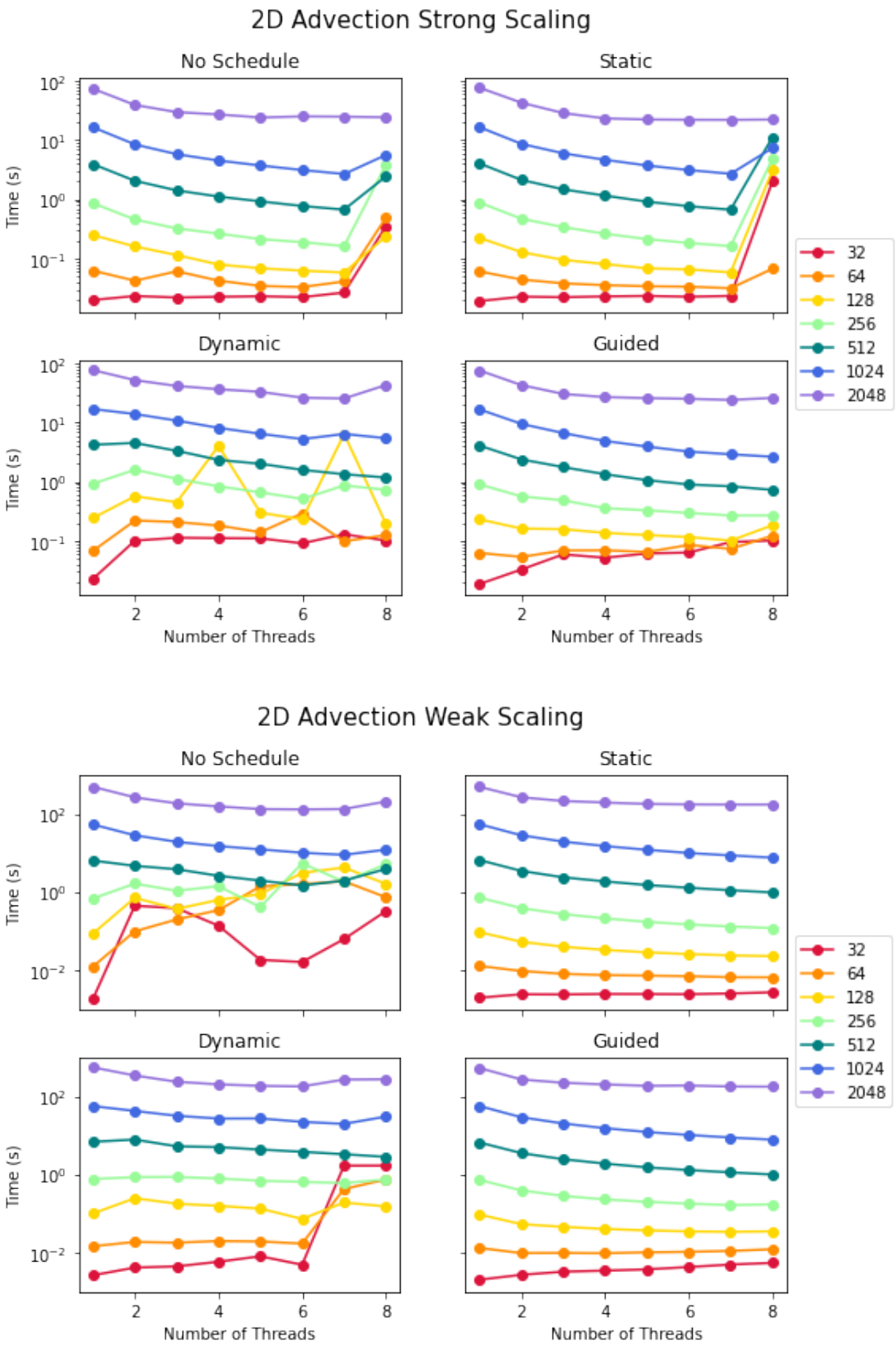


Figure 5: Strong and weak scaling for the 2D Advection Equation

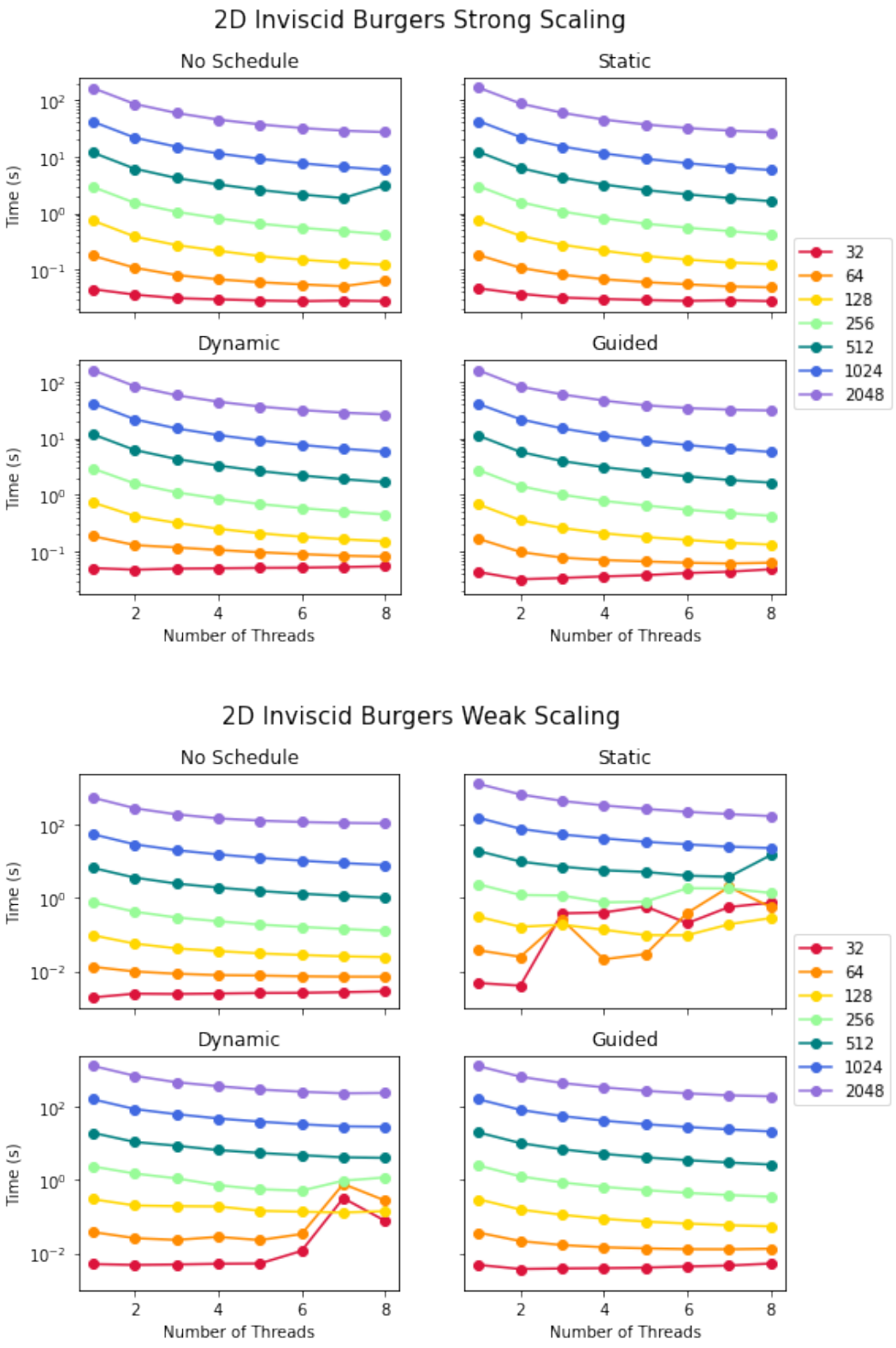


Figure 6: Strong and weak scaling for the 2D Inviscid Burgers' Equation

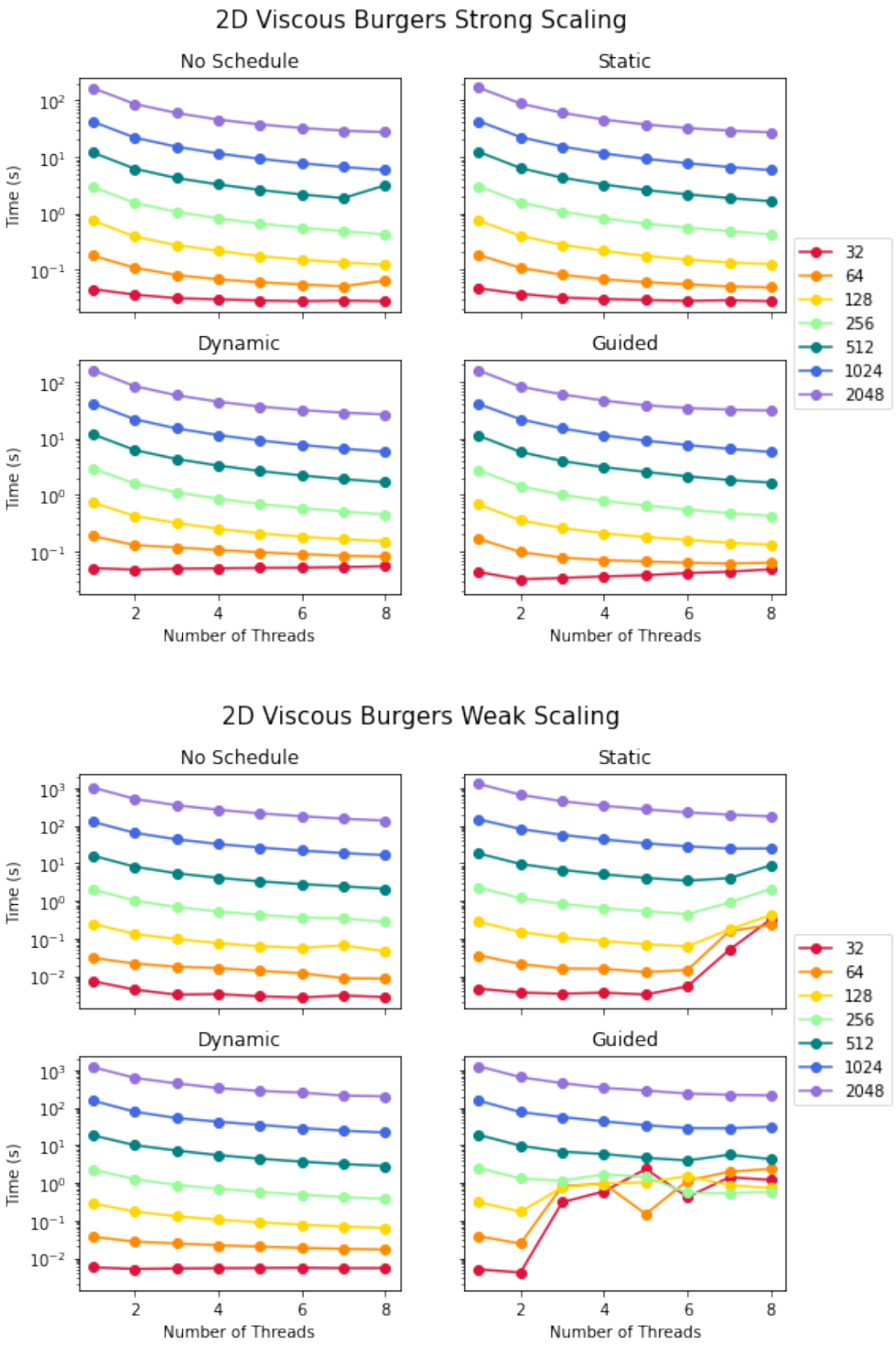


Figure 7: Strong and weak scaling for the 2D Viscous Burgers' Equation

For the 2D Inviscid and Viscous Burgers' Equations shown in Figures 9 and 10, no scheduler is clearly best for the case of weak scaling. There is also little-to-no variation between the other schedulers in terms of strong scaling. For all four cases, there is a general trend of increasing the thread count decreasing the computation time.

The overarching question of this research project is "How can I make my code run faster?" To that, we answer, more threads, no scheduler.

4 Acknowledgments

C. W. would like to thank Dr. Andrew Christlieb for his invaluable knowledge on numerical solutions to hyperbolic PDEs. C. W. and G. A. would both like to thank Dr. Bill Punch for teaching us how to run our code faster.

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.

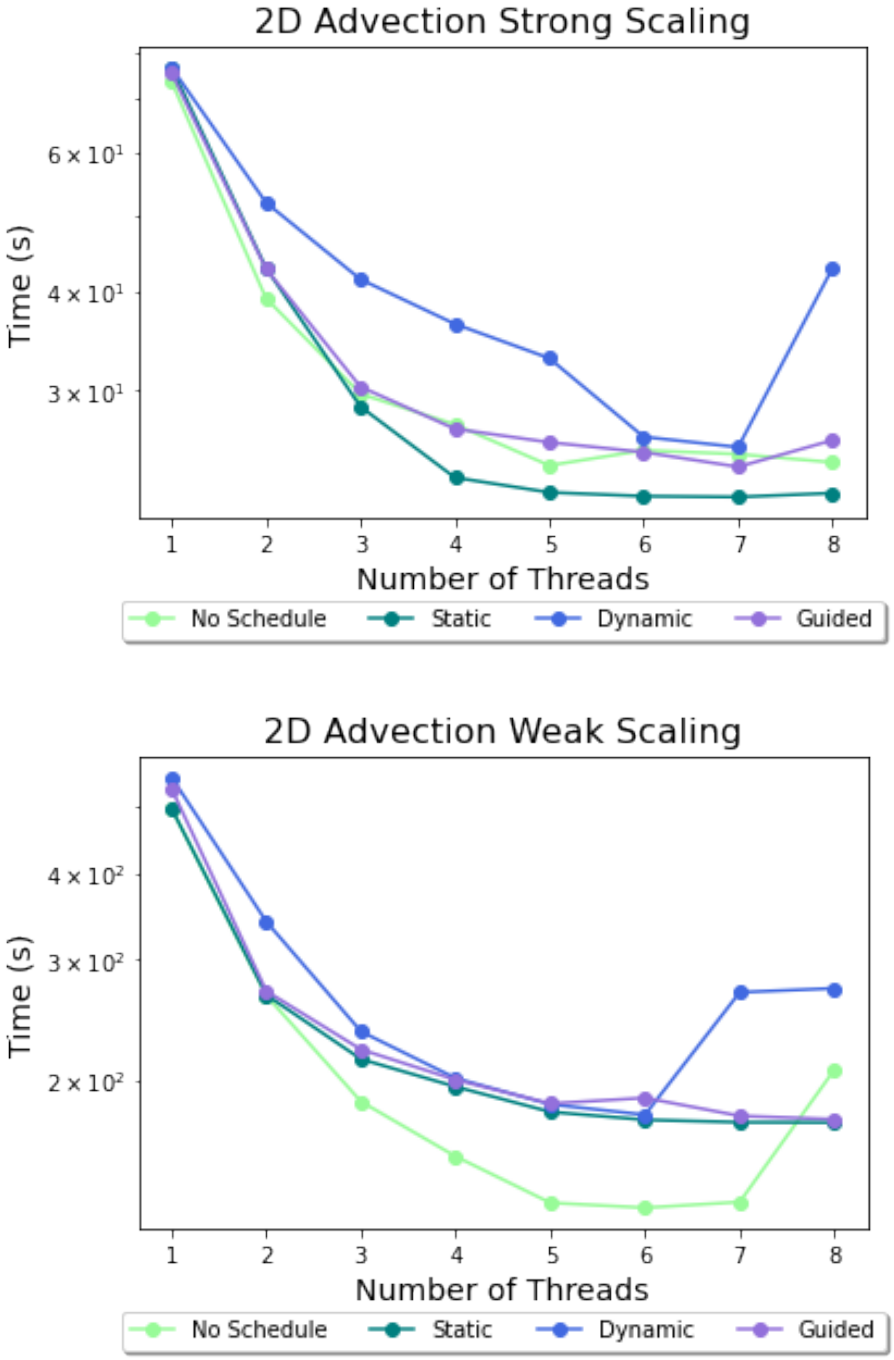


Figure 8: 2D Advection Equation

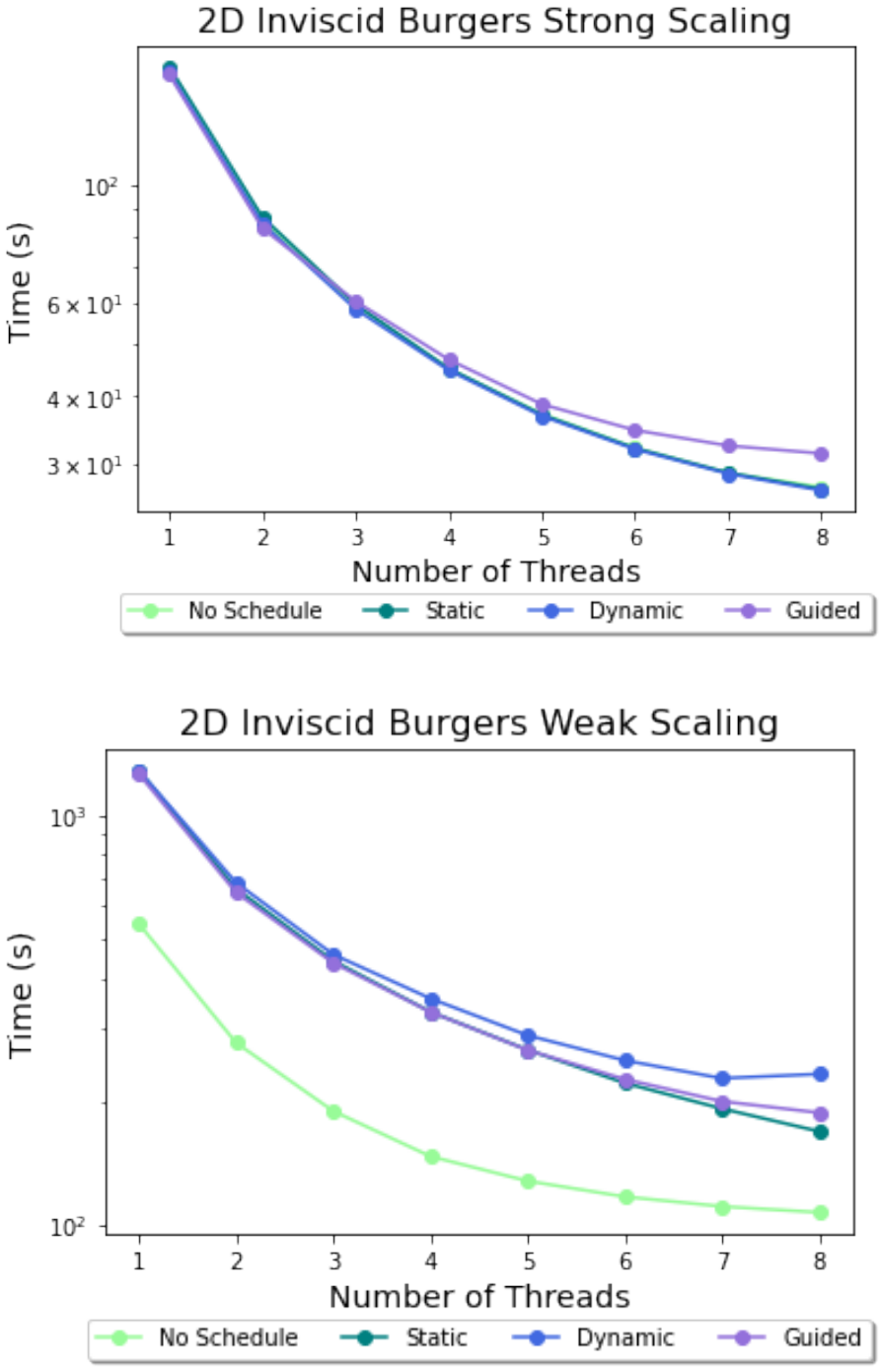


Figure 9: 2D Inviscid Burgers' Equation

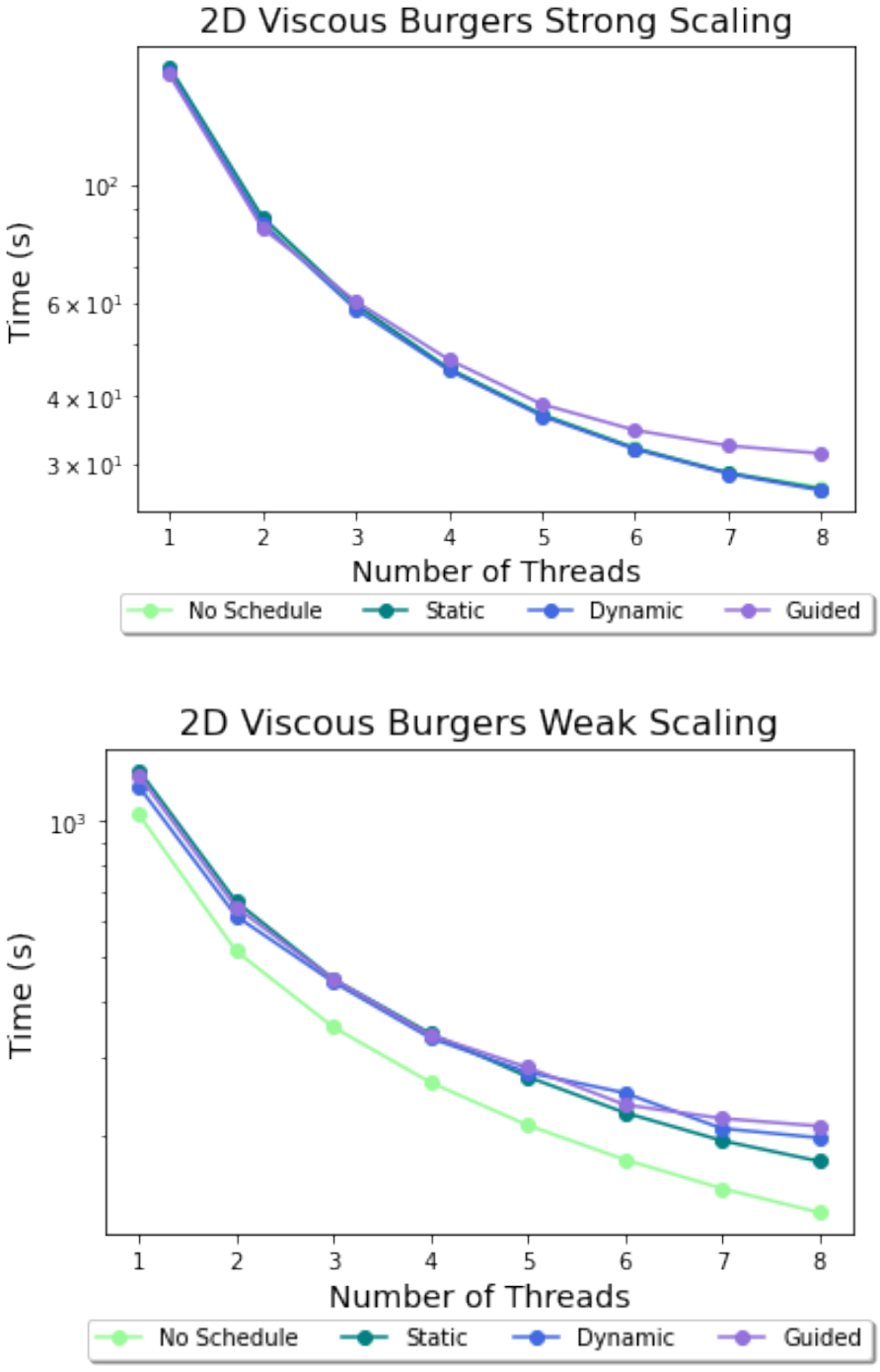


Figure 10: 2D Viscous Burgers' Equation