JULIATO: A JULIA-BASED PLASMA PHYSICS CODE WITH SERIAL OPTIMIZATIONS

VINCENT FAN , LUKA GOVEDIC , LUCAS SHOJI , AND ALEX VELBERG

Abstract. In this article, we report about Juliato, a Julia version of the plasma physics code Viriato with improvements. We analyze the Fourier-Hermite semi-implicit time integration approach and its variable dependency graph, finding properties that allow further parallelism and cache-locality. An algorithm is devised to take advantage of these properties. A serial version of the code is implemented and benchmarked against the original. Further optimization approaches are explored.

1. Introduction. Plasma simulations are relevant in a variety of situations. A great portion of the universe is composed of plasma, and modeling of this state of matter is essential to studies of phenomena including planetary magnetospheres, stars, and accretion disks around compact objects such as black holes. Plasmas are also the medium in which fusion reactors would operate, and understanding its turbulent transport of energy is fundamental to the success of the field. This state of matter is also critical to predicting space weather, essentially the changes in the solar wind that affect the operation of satellites and telecommunication on Earth.

Although useful in many situations, modeling plasmas has been a notoriously difficult task and is one of the most multi-scale problems in physics. This means that a wide separation between the small and large scales is needed inside the simulations. For example, to accurately simulate the environments in fusion reactor scales, one needs to resolve kinetic motion in scales around 10^5 smaller in space resolution. A similar situation occurs for the time axis. Taking the cube for the number of cells, it turns impossible even for the best high-performance systems to simulate it all. This multi-scale behavior emerges from the highly nonlinear, chaotic nature of plasma physics – fluctuations at the smallest scales is linked with changes in macroscopic behavior.

One approach to make plasma simulation more viable is to take velocity moments g_m of the relevant dynamical equations. This way, we can reduce the dimensionality of the equations to a set of coupled moment equations that are evolved in time. The original equation can be expanded into an indefinite number of moments, and a closure is required. This is usually done by taking all moments after some number n to zero, e.g. $g_m = 0$ for $m > n$. The more moments are used, more we can resolve fluctuations in velocity space.

Viriato [3] is a computational algorithm, originally implemented in Fortran, that uses such moments to simulate strongly magnetized plasma dynamics. It uses FFTs to compute spatial derivatives and performs dynamic time integration, changing the discrete integration step Δt depending on previous values. Viriato is pseudo-spectral, performing most operations in Fourier space while computing nonlinearities in real space. The code also uses a custom semi-implicit scheme that carries a complex dependency graph. The main task of this project is to build on Viriato's algorithm and improve it, exploring further parallelism and performance opportunities. The first step is to translate the parallel Fortran implementation into a serial Julia version, allowing for a deeper understanding of the code, as well as reducing the complexity associated with Viriato's parallelism. The serial Julia implementation, called "Juliato", will be used as a platform for future performance enhancements.

In section 2, we introduce the relevant plasma physics background and the Hermite moment approach. In section 3, we present the complex variable dependency

in Viriato's time integration, and discuss properties of this dependency graph that could be explored for better performance in 4. Next, we describe our implementation of a Julia version of Viriato and further serial optimizations to it in 5, compare results and benchmark against the original Viriato code in 6, devise a new algorithm that takes advantage of the dependency scheme in 7 and finally list possible further optimizations in 8.

2. Plasma Physics Background. A purely kinetic, or first-principles description of a plasma requires self-consistently tracking the motion of charged particles in electromagnetic fields through 6D phase space (3D+3V). In practice, a statistical description of the plasma is described via the Vlasov-Maxwell system of equations,

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla f + q\left(E + \mathbf{v} \times \mathbf{B}\right) \cdot \nabla_v f = C(f) \tag{2.1}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{2.2}$$

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \frac{1}{c^2}\frac{\partial \mathbf{E}}{\partial t} \tag{2.3}$$

where $f = f(\mathbf{x}, \mathbf{v}, t)$ is the particle distribution function, whose integral over all space is equal to the number density of the plasma, $\mathbf{v}$ is the velocity, q is the particle charge, $\mathbf{E}$ is the electric field, $\mathbf{B}$ is the magnetic field, ∇_v is the gradient operator with respect to velocity, and $C(f)$ is the collision operator, which encodes collisional effects such as viscosity and resistivity. Note that equation 2.1 must be solved for each plasma species (e.g. electrons and ions).

In principle, these equations can be discretized and time-integrated over a numerical grid, however, the difficulty lies in accurately resolving the physics. Many problems in plasma physics are inherently multi-scale, meaning that several physically important temporal and spatial scales must be accurately resolved by the discretization. For example, a simulation of a plasma in a fusion device may need to accurately resolve device-scale behavior on the order of 1 meter while simultaneously resolving the effects of turbulence at scales near the ion and electron gyroradii, ρ_i and ρ_e, on the order of 1 mm and 0.01 mm, respectively (these are the characteristic distance at which particles orbit magnetic field lines due to the Lorentz force). This will require a very large number of grid points, N, which in 3D will make our computation time scale as N^3. Including the scale separation required in velocity space and in time quickly results in an intractable computation.

To reduce dimensionality, a common technique is to take fluid moments of the distribution function via the operator,

$$\int_{-\infty}^{\infty} \mathbf{v}^m f d\mathbf{v} \tag{2.4}$$

where m represents the moment number. Applying this operator to equation 2.1 will result in an infinite hierarchy of PDEs for the evolution of the moments, where each equation for moment m depends on previous moments as well as the next highest moment, $m+1$. In the limit $m \to \infty$, this will exactly recover the distribution

function and the full velocity-space physics. There is therefore a trade-off between more accurately resolving velocity-space dynamics by including more moments and reducing simulation cost by truncating the hierarchy.

While low-moment approximations can be useful, we often want to refine our description of the plasma to include kinetic effects. For instance, the description of wave-particle interactions in the plasma via Landau damping is important for capturing kinetic plasma instabilities and energy dissipation channels. While it is possible to do this refinement via equation 2.4, there is no generalized form for the m-th PDE, so this requires deriving and hard-coding each unique PDE. For mathematical convenience, it is instead possible to use a Hermite moment expansion [2],

$$g_m(x,t) = \int_{-\infty}^{\infty} H_m(v) f d\mathbf{v} \tag{2.5}$$

where H_m is the m-th Hermite polynomial. These Hermite moments span the same space as those generated by 2.4, preserve the useful coupling between each m and its neighbors $m-1$ and $m+1$, and allow us to write a generalized equation for any moment. By leveraging this expansion and making some physics-informed reductions in the number of velocity dimensions (often to 2D: $v_\parallel$ and $\mathbf{v}_\perp$ with respect to the magnetic field), it is possible to reduce the computational cost of plasma physics simulations to a feasible level for modern computing clusters.

3. Performance issues in Viriato. Viriato [3] is a code that implements this Hermite scheme to evolve a reduced, 4D form of the Vlasov equations (3D+1V). Using generalized operators ($\mathcal{G}_m$) to represent all nonlinear components of the PDEs, the equations for the moments look as follows:

$$\frac{\partial g_m}{\partial t} = \mathcal{G}_m(g_0, g_1, g_{m-1}, g_m, g_{m+1}) - m\nu_{ei} g_m \tag{3.1}$$

where ν_{ei} is a constant. As discussed in the previous section, we see that the evolution of each g_m depends on its neighbor moments, g_{m-1} and g_{m+1}, as well as the first two moments g_0 and g_1 For g_0, this dependency reduces to only g_0 itself and g_1. An source of complexity is the nonlinearity of the $\mathcal{G}_m$ operators. These operators contain many Poisson brackets, defined as

$$[P,Q] \equiv \partial_x P \partial_y Q - \partial_y P \partial_x Q$$

This structure involving both spatial derivatives and multiplication requires significant computation and is the main hurdle for evolving the set of equations in 3.1. Viriato employs a pseudo-spectral scheme for calculating these brackets. To facilitate taking derivatives, the code works with variables in Fourier (spectral) space. However, spectral space makes array multiplications costly as they would involve convolutions. To avoid this problem within the brackets, the variables are transformed to real space for straightforward multiplication and transformed back to Fourier space soon after. The Fast Fourier Transform (FFT) calls in both directions end up being very costly for this algorithm, and tend to dominate asymptotic computation.

3.1. Semi-implicit time integration. To evolve the moments, the code performs discrete time integration through a semi-implicit scheme. This approach is taken to ensure numerical stability, evolving the stiff part of the equations implicitly and the non-stiff parts explicitly to avoid large matrix inversions.

First, a predictor step is taken to calculate a preliminary value of each $g_m^{n+1,*}$ moment from the values g_m^n from the previous time step:

$$g_m^{n+1,*} = e^{-m\nu_{ei}\Delta t} g_m^n + \frac{\Delta t}{2}\left(1 + e^{-m\nu_{ei}\Delta t}\right)\mathcal{G}_m(g_0^n, g_1^n, g_{m-1}^n, g_m^n, g_{m+1}^n) \tag{3.2}$$

Notice that this step depends on the variables g_0^n, g_1^n, g_m^n and the neighbors from the previous time step. After this, multiple iterations of a corrector step are taken until converge within a desired accuracy is achieved. Adopting the initial predictor value as the 0th step, the value for the $(p+1)$-th corrector step $g_m^{n,p+1}$ is:

$$\begin{aligned} g_m^{n+1,p+1} = e^{-m\nu_{ei}\Delta t} g_m^n + \frac{\Delta t}{2} e^{-m\nu_{ei}\Delta t}\mathcal{G}_m(g_0^n, g_1^n, g_{m-1}^n, g_m^n, g_{m+1}^n) + \\ \frac{\Delta t}{2} e^{-m\nu_{ei}\Delta t}\mathcal{G}_m(g_0^{n+1,p+1}, g_1^{n+1,p+1}, g_{m-1}^{n+1,p+1}, g_m^{n+1,p}, g_{m+1}^{n,p}) \end{aligned} \tag{3.3}$$

Notice the complicated dependency structure of this equation. Beyond using variables from the previous time step and the previous iteration (p) of the corrector step, it also depends on values of g_0, g_1 and g_{m-1} of the same $(p+1)$th iteration. In the special case for g_1, the equation does not depend on g_0^{p+1} from the same corrector step; It can be calculated directly with values from just the previous step. Thus, this moment can be calculated first to obtain all the others. We make a dependency chart for this predictor-corrector scheme for the case of one corrector step in figure 1.

Another relevant aspect of this dependency is its dynamic time-stepping scheme: each time interval Δt is determined by values of g_m in the previous time step. More specifically, the CFL condition is evaluated at each timestep in conjunction with the maximum value of g_1. The timestep is chosen based on either the maximum allowed by the CFL coefficient or an imposed relative error between steps based on g_1, whichever is smaller. Due to all this complexity, even though it is parallel in space Viriato has poor parallel scalability which will be discussed further in section 6.

4. Opportunities for parallelism and cache-locality. Despite the complicated dependency graph, there are opportunities for parallelism. There is also significant reuse, which could be exploited to increase the cache locality. These are the main *valuable properties*:

1. Ignoring the dependence on g_0 (and g_1), value $g_{m_1}^{t_1}$ does not depend on $g_{m_2}^{t_2}$ if $(m_2 - m_1) > 2 \cdot (t_1 - t_2)$. In other words, a moment m_1 at time t_1 does not depend on a higher moment m_2 at an earlier time t_2 if m_2 is much further away in space than it is in time. Let's call this t-parallelism.
2. Each g_m value is used between 2-4 times. For example, a predictor step $g_m^{n,*}$ value is used to compute $g_{m-1}^{n+1,p=1}$ and $g_m^{n+1,p=1}$, while a corrector step value g_m^{n+1} is used to compute $g_{m-1}^{n+2,*}$, $g_m^{n+2,*}$, $g_{m+1}^{n+2,*}$, and g_{m+1}^{n+1}.
3. Each $g_1(x, y)$ value will be used for computing g_m for all m in that time step for a given x, y. Let's call this m-reuse.

These *valuable properties* can be exploited by changing the high-level order of computation. However, there are more that we can exploit regardless of the high-level approach we take:

1. After the forward FFT to compute the bracket, all computation in xy-space is independent and can be done in parallel.

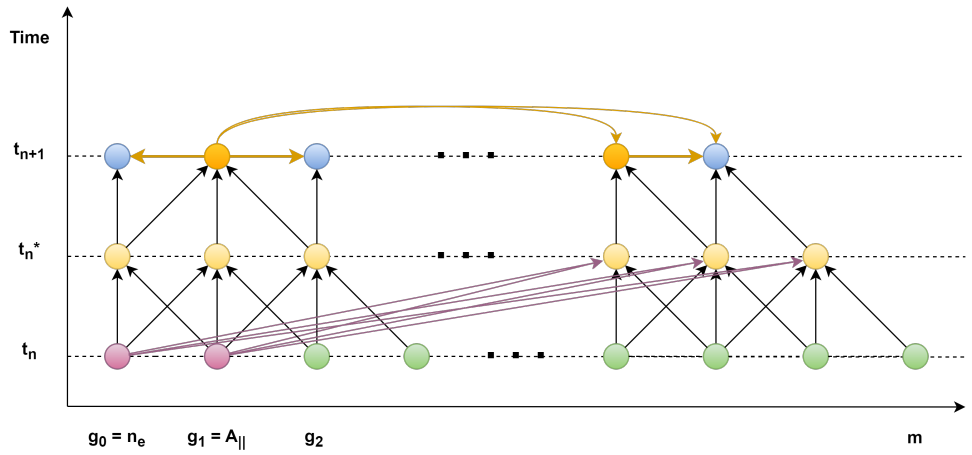


Fig. 1: Dependency diagram for the semi-implicit scheme, depicted as balls in m, t-space. Predictor step values (in yellow) only depend on g_0, g_1, and neighboring g_m from the already computed previous time step (in green). Corrector step values (blue and orange) depend on neighboring moments from the predictor step (yellow), as well as g_0, g_1, and g_{m-1} from the same corrector step, shown in orange. Finally, g_1 is computed first and used in the computation of g_0. That dependency is also shown in orange.

2. g_m will be stored as a multidimensional array in contiguous memory, so g_m values along the fastest-moving dimension will be on the same cache line. This allows for vectorization and spatial locality.

5. Implementation and Serial Optimization. We attach a high level overview of the Viriato algorithm in 5.1 written in pseudocode. The algorithm was originally implemented in Fortran, and a majority of this project focused on re-implementing the algorithm in Julia and C++ and then investigating some performance improvements.

Algorithm 5.1 Viriato

procedure NAIVE
  for $t \leftarrow 0, t_{\max}$ **do** ▷ Begin predictor step
    if $t = 0$ **then**
      Initialize
    end if
    for $m \leftarrow 0, M$ **do**
      $\widetilde{Br}_1 \leftarrow \text{BRACKET}(\widetilde{g}_m, \widetilde{g}_0)$
      $\widetilde{Br}_2 \leftarrow \text{BRACKET}(\widetilde{g}_{m-1} + \widetilde{g}_{m+1}, \widetilde{g}_1)$
      for $x \leftarrow 0, X; y \leftarrow 0, Y$ **do**
        $\widetilde{g}^*_m[x, y] \leftarrow \mathcal{G}_m(\widetilde{g}_m[x, y], \widetilde{Br}_1[x, y], \widetilde{Br}_2[x, y], DT)$
      end for
    end for
    for $m \leftarrow 0, M$ **do** ▷ Begin corrector step
      $\widetilde{Br}_1 \leftarrow \text{BRACKET}(\widetilde{g}^*_m, \widetilde{g}_0{}^{new})$
      $\widetilde{Br}_2 \leftarrow \text{BRACKET}(\widetilde{g}_{m-1}{}^{new} + \widetilde{g}^*_{m+1}, \widetilde{g}_1{}^{new})$

for $x \leftarrow 0, X; y \leftarrow 0, Y$ **do**
$\widetilde{g}_m^{new}[x,y] \leftarrow \mathcal{G}_m(\widetilde{g}_m[x,y], \widetilde{Br}_1[x,y], \widetilde{Br}_2[x,y], DT)$
end for
end for
Calculate RelError
if RelError $> \varepsilon$ **then**
$DT \leftarrow 0.92 \cdot DT$
Redo timestep
end if
ComputeNextDT($\widetilde{g}^{new}$)
Swap($\widetilde{g}, \widetilde{g}^{new}$)
end for
end procedure

procedure Bracket($\widetilde{v_1}, \widetilde{v_2}$)
$\delta_x v_1 \leftarrow FFT^{-1}(k_x \cdot \widetilde{v_1})$
$\delta_y v_2 \leftarrow FFT^{-1}(k_y \cdot \widetilde{v_2})$
for $x \leftarrow 0, X; y \leftarrow 0, Y$ **do**
$Br[x,y] \leftarrow \delta_x v_1[x,y]\delta_y v_2[x,y] - \delta_y v_1[x,y]\delta_x v_2[x,y]$
end for
$\widetilde{Br} \leftarrow FFT(Br)$
return $\widetilde{Br}$
end procedure

In the algorithm above, t represents the time parameter, where $t_{\max}$ determines the number of timesteps we would like to run the simulation for. M is the number of moments that we would like to resolve for, and X and Y are parameters that determine how fine the mesh grid we use is. DT is a parameter that determines the "physics" time step which the simulation reflects, and is calculated by the semi implicit scheme.

Viriato consists of an outer loop through all of the timesteps. If $t = 0$, then we initialize the moments depending on the input parameters. Each timestep consists of a predictor step and a corrector step.

In the predictor step, we use previous moments to compute the preliminary moments for the current time step, which consists of a large number of calls to bracket and various implementations of $\mathcal{G}_m$ (the implementation differs depending on m). In the corrector step, we use the preliminary values from the predictor step, as well as values of the moments from the previous time steps to re-calculate the moments for this time step. Finally, we compute the relative error. If the relative error is larger than some input threshold ε, we redo the entire timestep using a smaller value of DT.

For more details on Viriato, consult section 3 which describes the details each component of the implementation in more detail.

After the naive version of the algorithm was implemented in Julia, we sought to improve the performance first by making as many serial implementations as possible. In particular, we used `ProfileView.@profview` to profile the code in detail to observe which lines of code were taking up the most time.

Figure 2a shows a visualization of the stack trace for `viriato.jl` run with $X = 64, Y = 64, t_{\max} = 10$ and $M = 10$ implemented as the naive algorithm, and figure 2b shows the stack trace for `viriato.jl` after we implemented several serial

optimizations. The full timing results will be presented in the following section, but we achieved over 3x speedup by implementing these serial improvements. To interpret the stack trace visualizations, note that the x axis shows the relative time a certain line of code requires, and the y axis is essentially the stack trace of that line of code.

Since the two figures only show the relative timings of each line of code, it is not immediately clear that the second visualization represents a large improvement over the first visualization. However, note that the horizontal bands are generally wider in Figure 2b, which means that we were able to reduce the pieces of code that were taking up severely large proportions of time in our serial improvements. Also, the visualization is less "spiky", as we targeted the pieces of code that were often making too many calls to built in helper functions due to the nature of the original naive implementation. We describe the changes below.

Note that in our pseudocode, when we calculate the updated moments arrays, we loop through each index and assign it to a new value, since a new computation must be done at every position. This is similar to the actual implementation. One serial improvement we made was changing the order of the inner and outer loops. Since Julia is column major, we saw improvements in performance when we switched from `for i in 1:nkx, j in 1:nky` to `for j in 1:nky, i in 1:nkx` in all of the locations where we were iterating through two dimensional arrays. In general, switching to column major access patterns can improve cache efficiency, and also allow for the compiler to look for efficient vectorization opportunities. This was backed up by the `@benchmark` results we acquired immediately after only implementing this one improvement.

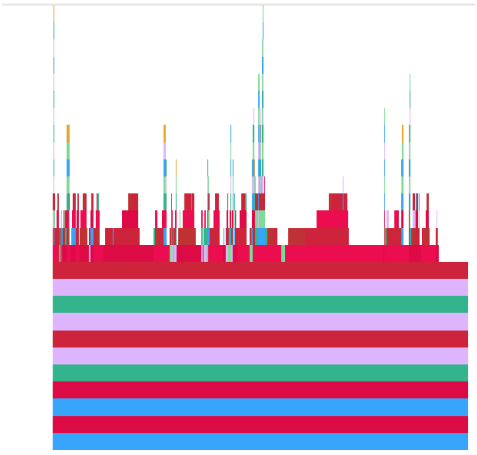
The next set of improvements came from observing that the stack trace made a large number of calls to helper functions that were computing various exponentials, such as `exp_nu` or `exp_eta` that were used in calculating the moments for the next timestep. We were able to improve performance by only calculating one value of each helper function with a loop step, and use the saved value for future calculations.

During the implementation process, we had uncovered an aliasing bug where the new set of values, such as for $A_{\|}$ would be assigned to an old variable such as in `akpar = akpar_new`, which actually caused both variable names to become references for the same object. We initially amended this issue by using `deepcopy`. During the serial improvements, we instead just switched the references of the two objects with `akpar, akpar_new = akpar_new, akpar`, which yielded small improvements in performance.

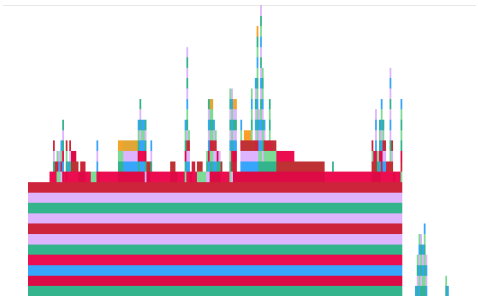
The largest improvements in performance came upon careful inspection of the stack trace and theorizing that our input parameters and constants could be loaded more efficiently. For one, we added `const` to every variable that would remain constant through the entire algorithm. Also, we found that we initially had far too many `include` statements, which we consolidated into only the file that contained `main()`.

6. Benchmarks and Serial Performance Results. We have performed benchmarks for Viriato and Juliato with and without serial improvements on MIT's Engaging cluster. Each node on Engaging is an Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz, containing 16 2-way hyperthreaded cores. Both Viriato's serial and parallel performance are benchmarked by timing the execution of the primary time-loop (line 2 in Algorithm 5.1) for 50 time steps. For serial performance, the minimum of a set of seven timings was taken, and for the parallel performance, the median value was taken. The simulation parameters used for benchmarking are reported in Table 1.

The results of benchmarking Viriato's parallel speedup are shown in Figure 3, where the poor performance compared to a linear speedup indicates ample oppor-



(a) Before serial improvements



(b) After serial improvements

Fig. 2: Stack trace visualizations of `Juliato` with $X = Y = 64$, $M = 10$ and $t_{\max} = 10$

Parameter	**Value**
`lx`, `ly`	1.0
`nlx`, `nly`	128
η, ν	0.1
`ngtot`	20

Table 1: Simulation Parameters used for all Benchmarks. `lx` and `ly` are the box length, `nlx` and `nly` are the number of grid points in the x and y directions, resistivity and viscosity are η and ν, and `ngtot` is the number of Hermite moments, M, not including 0 and 1.

tunities for improvement. On 16 cores, it only achieves a parallel speedup of cca. 5.34.

Juliato's serial performance was benchmarked through timing the primary loop with `BenchmarkTools.@benchmark`. Results are shown in Table 2.

Although both Julia versions remain slower than the original Fortran90 implementation, the serial optimizations to the Julia code described in section 5 resulted in a 3.3x speedup over our initial version. We ran out of time doing further performance engineering and investigation, but we suspect that the speed deficit of Juliato has to do with Julia being a higher-level language. While it allows you to write code that does not perform well (due to type-instability or garbage collection), it should be possible to achieve the full performance that Viriato achieves. For that, we'd need to take a look at the generated assembly code for both.

Aside from the performance benchmarks, it is also important that our code produces accurate physics results. We have performed some preliminary tests which compare the output of the 1st moment, the vector potential parallel to the magnetic field, $A_{\parallel}$ for the Orszag-Tang vortex initial condition, a commonly used setup for benchmarking plasma codes [3]. Both codes are run serially with the same input

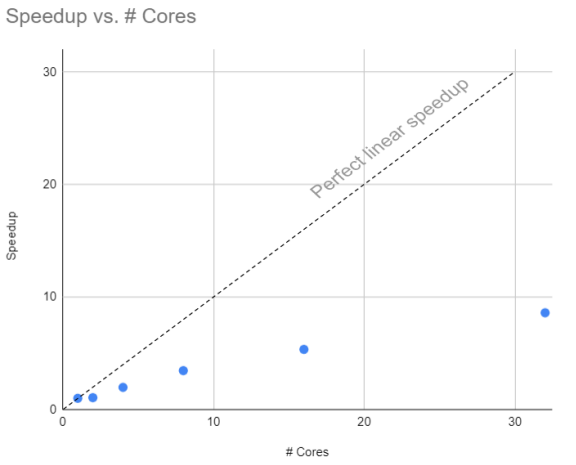


Fig. 3: Parallel speedup of Viriato on a Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz, which has 16 cores that are 2-way hyperthreaded. The speedup is measured compared to the running time on one core (larger is better). The dashed line shows perfect linear speedup.

Implementation	**Normalized Compute Time**
Viriato	1.0
Juliato (initial)	10.3
Juliato (serial optimization)	3.1

Table 2: Serial performance normalized to Viriato serial performance using parameters given in 1.

parameters and the same output cadence (every 3200 timesteps). If Juliato perfectly replicates the values in Viriato, we would expect identical results at a given time-loop iteration, regardless of performance. The simplest way to test this is to compare the physics time achieved by each simulation at a given time iteration. Since the code adaptively changes the physics timestep to meet the CFL condition (it restricts the propagation of a wave to less than 1 grid point per timestep based on the wave velocity), the dynamics of the problem will modify how big the timesteps are for a given time iteration. A comparison between Viriato and Julia is shown in Figure 4. While there is some deviation over time, this does not reveal any major issues with Juliato. More careful benchmarking in the future will aim to resolve this discrepancy.

Figure 5 shows a colormap of $A_{\|}$ at time iterations 3200 and 6400, as well as the absolute value of the relative error. We note qualitative agreement in the dynamics, but due to the discrepancy observed in 4, it is difficult to make a direct comparison via a relative error calculation–the errors are influenced by the fact that the simulation time is different in the comparison.

7. A better approach: SkewedTableau. A SkewedTableau algorithm adopts a less straightforward order of computation to better exploit *valuable properties*

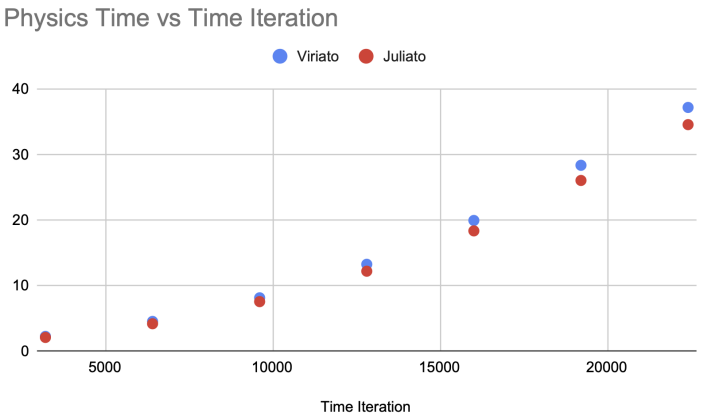


Fig. 4: Simulation time deviation between Viriato and Juliato over 22,400 timesteps.

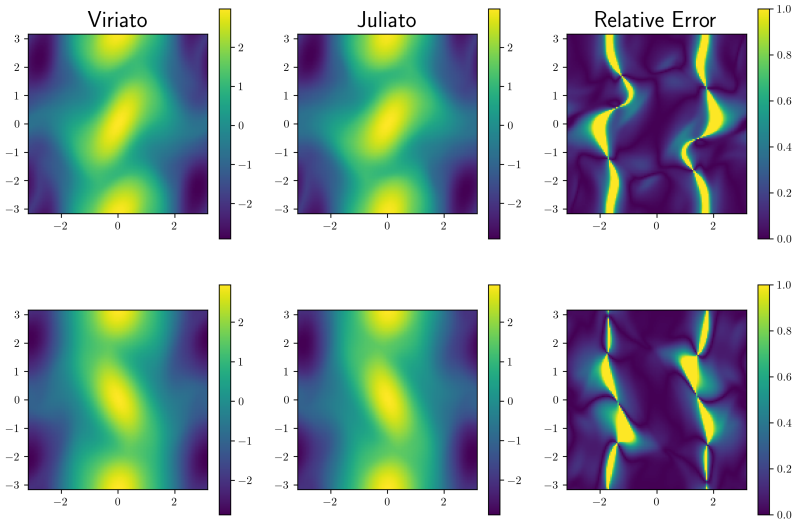


Fig. 5: Comparison of $A_{\|}$ from Viriato and Juliato at 3200 (first row) and 6400 (second row) time iterations, as well as the relative error.

introduced in section 4. It is similar to a parallel recursive solution of a simple tableau computation problem; both are shown in Figure 6. Because SkewedTableau is not balanced (the top-left yellow region is smaller than the bottom right), low-overhead load balancing is required, which OpenCilk can provide.

Additional complexity arises here because we suddenly need to store values of g for more than just one timestep. That is especially true for g_0 and g_1, which are

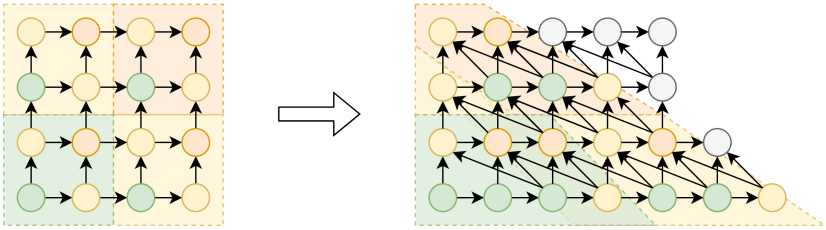


Fig. 6: The simple tableau construction (left) and our dependency scheme (right), along with their parallel recursive implementations. On the left, we recursively divide the tableau into 4 subregions, and compute the green one first, then the two yellow ones in parallel, and finally the orange one. Inside each region, we recursively repeat each process. On the right, although complex, we can do something similar. The predictor step is omitted to reduce the visual load.

always needed. However, if memory usage becomes a problem, we can always allocate the memory on each worker when it's needed, which will bound it by P-times the minimum memory required during serial execution. It is important to note that the SkewedTableau algorithm parallelizes over Hermite-moment-space, an optimization not found in Viriato, which only parallelizes by tiling the physical volume. Implementing the SkewedTableau in conjunction with parallelization across the physical volume could greatly speed up computation compared to Viriato when a high number of moments are required, as is the case for the highest fidelity simulations.

8. Further Optimizations. This section lists possible low-level optimizations that might improve the performance of either approach. After looking at preliminary benchmarking results, some optimizations do not seem worth it while others became more interesting.

8.1. Manual Vectorization. Despite only $O(1)$ arithmetic operations per memory operation, the time integration step is non-trivial and is quite compute-intensive. In the C++ version, the compiler does not currently fully vectorize the computation, and it does not use the full AVX512 vector lane width. According to preliminary profiling results, a large chunk of the time is spent inside the inner loop that computes the moment values, so getting that loop to vectorize more aggressively should result in further performance gains.

8.2. Data Layout. While it was initially suspected that experimenting with different data layouts might prove beneficial, due to FFT being the bottleneck, it was decided to keep x and y as the fastest-running dimensions, so that contiguous chunks can be passed to FFTW.

8.3. FFT. FFT remains one of the bottlenecks of the computation due to its $O(n \log n)$ complexity, even when using FFTW [1]. However, preliminary results show that executing multiple transforms at a time using library capabilities does not benefit performance. Additionally, bracket computation repeatedly allocates and deallocates temporary buffers, but the amount of time needed for that is pretty much negligible.

While FFT implementations achieve polylogarithmic span, they do not scale well

in practice[1]. Because library implementations use custom thread scheduling, they are incompatible with the OpenCilk scheduler. A benchmark that tested the scalability of running multiple FFTs in parallel was written, and it turned out well, almost achieving perfect linear speedup, increasing hope about parallelizing the computation on a high level.

9. Conclusions. The main product of this project is the Julia implementation of Viriato. Although it is not as fast as the Fortran version Juliato can achieve qualitatively similar physics results. As a serial implementation, Juliato unpacks some of the complexity in Viriato's MPI parallelization and will serve as a platform for further serial improvements and the implementation of the cache-localized parallel algorithm developed for this project. Although we could not implement the SkewedTableau algorithm in time for this semester and further verification will be needed before Juliato is ready for production use, we hope to continue development to speed up real research tasks. Throughout this project, we have learned a lot about physics simulation codes, parallel computing, and optimization in Julia. It was also rewarding to work on an interdisciplinary project, combining plasma physics and algorithm performance engineering.

Acknowledgments. We would like to thank Prof. Nuno Loureiro, Prof. Rezaul Chowdhury, and Dr. Alexandros-Stavros Iliopoulos for extremely helpful discussions and support throughout this project.

GitHub Repository. The GitHub repository for this project is open-source and can be found at the following link:
https://github.com/Ozymandias314/adaptive-hermite-refinement

REFERENCES

[1] M. Frigo and S. Johnson, *The design and implementation of fftw3*, Proceedings of the IEEE, 93 (2005), pp. 216–231, https://doi.org/10.1109/JPROC.2004.840301.

[2] G. W. Hammett, M. A. Beer, W. Dorland, S. C. Cowley, and S. A. Smith, *Developments in the gyrofluid approach to tokamak turbulence simulations*, Plasma Physics and Controlled Fusion, 35 (1993), p. 973, https://doi.org/10.1088/0741-3335/35/8/006, https://dx.doi.org/10.1088/0741-3335/35/8/006.

[3] N. F. Loureiro, W. Dorland, L. Fazendeiro, A. Kanekar, A. Mallet, M. S. Vilelas, and A. Zocco, *Viriato: a Fourier-Hermite spectral code for strongly magnetised fluid-kinetic plasma dynamics*, Computer Physics Communications, 206 (2016), pp. 45–63, https://doi.org/10.1016/j.cpc.2016.05.004.

[1] According to Prof. Chowdhury. We searched, but could not find references that support this. Publicly available FFT benchmarks also do not evaluate the libraries on more than one core.